django-getpaid Documentation

Release 1.7.0

Krzysztof Dorosz

Jul 25, 2018

Contents

django-getpaid is a carefully designed multi-broker payment processor for django applications. The main advantages of this dja

- instead of only one payment broker, you can use **multiple payment brokers** in your application, what is wise considering ane single payment broker downtime,
- payment brokers have a **flexible architecture** each one based on a django application structure (they can introduce own logic, views, urls, models),
- support to **asynchronous** payment status change workflow (which is required by most brokers and is architecturally correct for production use),
- support for payments in **multiple currencies** at the same time,
- uses just a **minimal assumption** on your code, that you will have some kind of order model class.

The basic usage is to connect you order model class with *django-getpaid*. Because the *AbstractMixin* is used, Payment model class uses a real `ForeignKey` to your order class model, so it avoids messy django `content_type` relations.

This app was written because there still is not a single reliable or simple to use payment processor. There are many payments related projects out there like Satchmo, python-payflowpro, django-authorizenet, mamona, django-paypal, django-payme, but none of them are satisfying. *Mamona* project was the most interesting payment app out there (because of general conception), but still has got some serious architectural pitfalls. Therefore *django-getpaid* in the basic stage was aimed to be a next version of *mamona*. *django-getpaid* took many architectural decisions different than *mamona* and therefore has been started as a separate project, while still borrowing a lot of great ideas from *mamona*, like e.g. using *AbstractMixin*, dynamic model and urls loading, etc. Thanks *mamona*!

Disclaimer: this project has nothing in common with getpaid plone project. It is mostly based on mamona project.

CHAPTER 1

Contents:

1.1 Installation

1.1.1 Gettings source code

The source code of the first stable version will be available on pypi.

For now you can download development version from: https://github.com/cypreess/django-getpaid.git

Installing development version can be done using pip:

```
$ pip install -e git+https://github.com/cypreess/django-getpaid.git#egg=django-getpaid
```

1.1.2 Enabling django application

Second step is to enable this django application by adding it to `INSTALLED_APPS` in your `settings.py`:

```
INSTALLED_APPS += ('getpaid', )
```

and add getpaid to your project's urls.py:

```
url(r'^getpaid/', include('getpaid.urls', namespace='getpaid', app_name='getpaid')),
```

Warning: It is advised that you pass `namespace` and `app_name` as kwargs to include.

1.1.3 Enabling getpaid backends

Create any iterable (tuple or list) named `GETPAID_BACKENDS` and provide full path for backends that you want to enable in your project. E.g.:

```
GETPAID_BACKENDS = ('getpaid.backends.dummy',
                    'getpaid.backends.payu', )
```

After that put also in your `settings.py` a dictionary `GETPAID_BACKENDS_SETTINGS`. This will keep all configurations specific to a single backend. Keys of this dictionary should be a full path of getpaid backends. Please refer to *Payment backends* section for names of available backends.

Each key should provide another dictonary object with some set of `key->value` pairs that are actual configuration settings for given backend:

```
GETPAID_BACKENDS_SETTINGS = {
    # Please provide your settings for backends
    'getpaid.backends.payu' : {

        },
}
```

1.1.4 Tests

Some tests are provided in getpaid_test_project/tests.py as a django test suite. You can run them with:

```
$ ./manage.py test orders --settings=getpaid_test_project.settings_test
```

1.2 Settings

1.2.1 GETPAID_BACKENDS

Required

Iterable with fully qualified python importable names of backends.

Example:

```
GETPAID_BACKENDS = ('getpaid.backends.dummy',
                    'getpaid.backends.payu', )
```

Note: The very common thing what are also should do, never to forget adding backends also to `INSTALLED_APPS`, is to add following line:

```
INSTALLED_APPS += GETPAID_BACKENDS
```

1.2.2 GETPAID_BACKENDS_SETTINGS

Optional

Dict that keeps backend specific configurations. Keys of this dictionary should be a fully qualified path of getpaid backends. Each key should be another dict with a backed specific keys. Please referrer to *Payment backends* section for names of available backends and theirs specific required keys.

Default: `{}`

Example:

```
GETPAID_BACKENDS_SETTINGS = {
    'getpaid.backends.transferuj' : {
            'id' : 123456,
            'key' : 'xxxxxxxxxxxx',
            'signing' : True,       # optional
        },
}
```

Warning: In spite of the fact that this setting is optional (e.g. not needed if only the dummy backend is used) every real backend requires some additional configuration, and will raise ImproperlyConfigured if required values are not provided.

1.2.3 GETPAID_ORDER_DESCRIPTION

Optional

String in Django template format used to render name of order submitted to payment broker. If the value is omitted or it evals to False, unicode representation of Order object will be used.

The following context variables are available inside the template:

order order object

payment payment object

Example:

```
GETPAID_ORDER_DESCRIPTION = "Order {{ order.id }} - {{ order.name }}"
```

Note: Setting this value has sense only if you are going to make `Order.__unicode__()` very custom, not suitable for presenting to user. Usually you should just define `__unicode__` method on your `Order` object and use it everywhere in the system.

1.2.4 GETPAID_ORDER_MODEL

Optional String describing model name.

Example:

```
GETPAID_ORDER_MODEL = 'my_super_app.Order'
```

Note: Required for django >=1.7

1.2.5 GETPAID_SUCCESS_URL_NAME

Optional Success URL name where the payment backend should return to on success

Example:

```
GETPAID_SUCCESS_URL_NAME = 'order_payment_success'
```

1.2.6 GETPAID_FAILURE_URL_NAME

Optional Success URL name where the payment backend should return to on failure

Example:

```
GETPAID_FAILURE_URL_NAME = 'order_payment_failure'
```

1.3 Payment workflow integration

With a few simple steps you will easily integrate your project with django-getpaid. This module is shipped with a very well documented django-getpaid test project which is packed together with the source code. Please refer to this code for implementation details.

1.3.1 Connect urls

Required

Add to your urls:

```
url(r'', include('getpaid.urls')),
```

1.3.2 Preparing your order model

Required

First of all you need a model that will represent an order in you application. It does not matter how complicated the model is or what fields it provides, if it is a single item order or multiple items order. You can also use a previously defined model you have, even if it's from a 3rd party app. Let's take an example from the test project:

```
from django.core.urlresolvers import reverse
from django.db import models
import getpaid

class Order(models.Model):
    name = models.CharField(max_length=100)
    total = models.DecimalField(decimal_places=2, max_digits=8, default=0)
    currency = models.CharField(max_length=3, default='EUR')
    status = models.CharField(max_length=1, blank=True, default='W', choices=(('W',
↪'Waiting for payment'),
                                                                               ('P',
↪'Payment complete')))
    def get_absolute_url(self):
        return reverse('order_detail', kwargs={'pk': self.pk})

    def __unicode__(self):
        return self.name

getpaid.register_to_payment(Order, unique=False, related_name='payments')
```

For django 1.8 please add the following line to your settings:

GETPAID_ORDER_MODEL = 'my_super_app.Order'

The class name is not important at all. Important is that you register your model using the `register_to_payment` method.

`getpaid.`**`register_to_payment`**(**args*, ***kwargs*)
Thin proxy for actual register_to_payment to prevent uncontrolled early loading of models directory.

You can add some *kwargs* that are basically used for `ForeignKey` kwargs. In this example whe allow of creating multiple payments for one order, and naming One-To-Many relation.

There are two important things on that model. In fact two methods are required to be present in order class. The first one is `__unicode__` method as this will be used in few places as a fallback for generating order description. The second one is `get_absolute_url` method which should return the URL from the order object. It is used again as a fallback for some final redirections after payment success or failure (if you do not provide otherwise).

It is also important to note that it actually doesn't mather if you store the order *total* in database. You can also calculate it manually, for example by summing the price of all items. You will see how in further sections.

Warning: Remember to run `./manage.py syncdb` in order to create additional database tables.

Controlling payment creation for an order

Getpaid supports payment creation policy for an order. It means that your order class can implement a method `is_ready_for_payment()` which will inform getpaid if the creation of a payment for the given order is allowed. This is a typical situation if e.g. you want to disallow to make another payment for an order that has the status "already paid" or that is expired by now. If you do not implement this method, getpaid will assume that paying this order is always allowed.

1.3.3 Preparing payment form for an order

Required

Your application after some custom workflow just created an order object. That's fine. We now want to get paid for that order. So let's take a look on a view for creating a payment for an order:

```
from django.views.generic.detail import DetailView
from getpaid.forms import PaymentMethodForm
from getpaid_test_project.orders.models import Order

class OrderView(DetailView):
    model=Order

    def get_context_data(self, **kwargs):
        context = super(OrderView, self).get_context_data(**kwargs)
        context['payment_form'] = PaymentMethodForm(self.object.currency, initial={
↪'order': self.object})
        return context
```

Here we get a `PaymentMethodForm` object, that is parametrised with the currency type. This is important, because this form will only display payment methods that accept the given currency.

`PaymentMethodForm` provides two fields: HiddenInput with order_id and ChoiceField with the backend name. This is how you use it in a template:

```
<form action="{% url 'getpaid:new-payment' currency=object.currency %}" method="post">
    {% csrf_token %}
    {{ payment_form.as_p }}
    <input type="submit" value="Continue">
</form>
```

The action URL of this form should point to the named url *getpaid:new-payment* that requires the currency code argument. This form will redirect the client from the order view directly to the page of the payment broker.

When client submits this form he will be redirected to getpaid internal view (`NewPaymentView`) which will do one of two things:

- redirect client to a payment broker (directly, via HTTP 302) - this action is made if payment backend is configured to contact with payment broker via GET method,

  Note: Using GET method is recommended as it involves less intermediate steps while creating a payment.

- display intermediate page with a form with external action attribute - this action is made if payment backend is configured to use POST method (or this is the only way to communicate with payment broker).

  Note: This intermediate page is displayed only to emulate making a POST request from the client side. getpaid displays a template `"getpaid/payment_post_form.html"` that should be definitely overridden in your project. On this page you should display some information *"please wait, you are being redirected to payment broker"* and add some JavaScript magic to submit the form automatically after page is loaded. The following context variables are available in this template:

  - `form` - a form with all input of type `hidden`,
  - `gateway_url` - an external URL that should be used in `action` attribute of `<form>`.

  This is an example of very basic template that could be used (assuming you are using jQuery):

```
<script>
    $(function(){
        $("#new_payment").submit();
    });
</script>
<p> Please wait, you are being redirected to payment broker </p>
<form action="{{ gateway_url }}" method="post" id="new_payment">
    {{ form.as_p }}
</form>
```

Warning: Do **not** put the `{% csrf %}` token in this form, as it will result in a CSRF leak. CSRF tokens are only used for internal URLs. For more detailed info please read Django CSRF Documentation.

Warning: Remember that using POST methods do not bring any significant security over GET. On the one hand using POST is more correct according to the HTTP specification for actions that have side effects (like creating new payment), on the other hand using GET redirects is far easier in this particular case and it will not involve using hacks like "auto submitting forms on client side". That is the reason why using GET to connect with the payment broker system is recommended over using POST.

1.3.4 Filling necessary payment data

Required

Because the idea of whole module is that it should be loosely coupled, there is this convention that it does not require any structure of your order model. But it still needs to know some transaction details of your order. For that django signals are used. djang-getpaid while generating gateway redirect url will emit to your application a `getpaid.signals.new_payment_query` signal. Here is the signal declaration:

```
new_payment_query = Signal(providing_args=['order', 'payment'])
new_payment_query.__doc__ = """
Sent to ask for filling Payment object with additional data:
    payment.amount:                 total amount of an order
    payment.currency:               amount currency
This data cannot be filled by ``getpaid`` because it is Order structure
agnostic. After filling values just return. Saving is done outside signal.
"""
```

Your code has to implement some signal listeners that will inform the payment object with the required information:

```
from getpaid import signals

def new_payment_query_listener(sender, order=None, payment=None, **kwargs):
    """
    Here we fill only two obligatory fields of payment, and leave signal handler
    """
    payment.amount = order.total
    payment.currency = order.currency

signals.new_payment_query.connect(new_payment_query_listener)
```

So this is a little piece of logic that you need to provide to map your order to a payment object. As you can see, you can do all fancy stuff here to get the order total value and currency code.

Note: If you don't know where to put your listeners code, we recommend to put it in `listeners.py` file and then add a line `import listeners` to the end of your `models.py` file. Both files (`listeners.py` and `models.py`) should be placed in one of your apps (possibly an app related to the order model).

Note: One may wonder why isn't this handled directly on the order model via methods like get_total() and get_currency(). It was a design consideration that you may not have access to your order model and therefore couldn't add these methods. By using signals, it does not matter if you have control or not over the order model.

Optional

Most likely you would also like to give some sort of information about your customer to your payment processor. The signal `getpaid.signals.user_data_query` fills this gap. Here is the declaration:

```
user_data_query = Signal(providing_args=['order', 'user_data'])
user_data_query.__doc__ = """
Sent to ask for filling user additional data:
    user_data['email']:             user email
    user_data['lang']:      lang code in ISO 2-char format
This data cannot be filled by ``getpaid`` because it is Order structure
agnostic. After filling values just do return.
"""
```

On the example above we are passing the customer email and its desired language. Some backends may also need additional information like the customers address, phone, etc.

1.3.5 Handling changes of payment status

Required

Signals are also used to inform you that some particular payment just changed status. In this case you will use `getpaid.signals.payment_status_changed` signal which is defined as:

```
payment_status_changed = Signal(providing_args=['old_status', 'new_status'])
payment_status_changed.__doc__ = """Sent when Payment status changes."""
```

example code that handles status changes:

```
from getpaid import signals

def payment_status_changed_listener(sender, instance, old_status, new_status,␣
↪**kwargs):
    """
    Here we will actually do something, when payment is accepted.
    E.g. lets change an order status.
    """
    if old_status != 'paid' and new_status == 'paid':
        # Ensures that we process order only once
        instance.order.status = 'P'
        instance.order.save()

signals.payment_status_changed.connect(payment_status_changed_listener)
```

For example, when the payment status changes to 'paid' status, this means that all necessary amount was verified by your payment broker. You have access to the order object at `payment.order`.

1.3.6 Handling new payment creation

Optional

For some reasons (e.g. for KPI benchmarking) it can be important to you to how many and which payments were made. For that reason you can handle `getpaid.signals.new_payment` signal defined as:

```
new_payment = Signal(providing_args=['order', 'payment'])
new_payment.__doc__ = """Sent after creating new payment."""
```

Note: This method will enable you to make on-line KPI processing. For batch processing you can just query a database for Payment model as well.

1.3.7 Setup your payment backends

Required

Please be sure to read carefully section *Payment backends* for information of how to configure particular backends. They will probably not work out of the box without providing some account keys or other credentials.

1.4 Payment backends

Payment backends are plug-and-play django applications that will make all necessary work to handle payment process via different money brokers. This way you can handle multiple payment providers (not only multiple payment methods!) what is wise taking into account possible problems like. breakdown or downtime of a single payment broker.

Each payment backend delivers a payment channel for a defined list of currencies. When using a `PaymentMethodForm` for displaying available payment methods for order it will automatically filter payment methods based on the provided payment currency. This is important as the system is designed to handle payments in multiple currencies. If you need to support currency conversions, you will need to do it before calling the payment method form in your part of application.

Just to have an overview, a payment backend has a very flexible architecture, allowing you to introduce your own logic, urls and even models.

Warning: Getting real client IP address from HTTP request meta

Many payment brokers for security reason require you to verify or provide real client IP address. For that reason the getpaid backend commonly uses `REMOTE_ADDR` HTTP meta. In most common production deployments your django app will stand after a number of proxy servers like Web server, caches, load balancers, you name it. This will cause that `REMOTE_ADDR` will almost **always** reflect the IP of your Web proxy server (e.g. 127.0.0.1 if everything is set up on local machine).

For that reason you need to take care by yourself of having set `REMOTE_ADDR` correctly so that it actually points to **real** client IP. A good way to do that is to use commonly used HTTP header called `X-Forwarded-For`. This header is set by most common Web proxy servers to the **real** client IP address. Using simple django middleware you can rewrite your request data to assure that **real** client IP address overwrites any address in `REMOTE_ADDR`. One of the solutions taken from The Django Book is to use following middleware class:

```
class SetRemoteAddrFromForwardedFor(object):
    def process_request(self, request):
        try:
            real_ip = request.META['HTTP_X_FORWARDED_FOR']
        except KeyError:
            pass
        else:
            # HTTP_X_FORWARDED_FOR can be a comma-separated list of IPs.
            # Take just the first one.
            real_ip = real_ip.split(",")[0]
            request.META['REMOTE_ADDR'] = real_ip
```

Enabling this middleware in your `settings.py` should fix the issue. Just make sure that your Web proxy server is actually setting `X-Forwarded-For` HTTP header.

Warning: Set Sites domain name

This module requires Sites framework to be enabled. All backends are based on Sites domain configuration (to generate a fully qualified URL for a payment broker service). Please be sure that you set a correct domain for your deployment before running `getpaid` or setup *GETPAID_SITE_DOMAIN* properly to point on the right domain (see *getpaid.utils.get_domain* for url fetching order).

1.4.1 Dummy backend `getpaid.backends.dummy`

This is a mock of payment backend that can be used only for testing/demonstrating purposes.

Dummy backend is accepting only `USD`, `EUR` or `PLN` currency transactions. After redirecting to a payment page it will display a dummy form where you can accept or decline a payment.

Enable backend

Add backend full path to `GETPAID_BACKENDS` setting:

```
GETPAID_BACKENDS += ('getpaid.backends.dummy', )
```

Don't forget to add backend path to `INSTALLED_APPS` as well:

```
INSTALLED_APPS += ('getpaid.backends.dummy', )
```

Setup backend

No additional setup is needed for the dummy backend.

1.4.2 PayU.pl backend `getpaid.backends.payu`

This backend can handle payment processing via Polish money broker PayU which is currently a part of the biggest e-commerce providers on Polish market - Allegro Group.

PayU accepts only payments in `PLN`.

Enable backend

Add backend full path to `GETPAID_BACKENDS` setting:

```
GETPAID_BACKENDS += ('getpaid.backends.payu', )
```

Don't forget to add backend path to `INSTALLED_APPS` also:

```
INSTALLED_APPS += ('getpaid.backends.payu', )
```

There is no need to add any url definitions to the main `urls.py` file, as they will be loaded automatically by `getpaid` application.

Setup backend

In order to start working with PayU you will need to have an activated account in PayU service. There you will need to define a new Shop with new Point of Sale (POS). This will give you access to following configuration variables:

pos_id POS identificator,

key1 according to PayU documentation this is a string that is used to compute md5 signature sent by Shop,

key2 according to PayU documentation this is a string that is used to compute md5 signature sent from Shop,

pos_auth_key just a kind of secret password for POS.

You need to provide this information in `GETPAID_BACKENDS_SETTINGS` dictionary:

```
GETPAID_BACKENDS_SETTINGS = {
    'getpaid.backends.payu' : {
            'pos_id' : 123456,
            'key1' : 'xxxxxxxxxxxx',
            'key2' : 'xxxxxxxxxxxx',
            'pos_auth_key': 'xxxxxxxxx',
            'signing' : True,       # optional
            'testing' : True,       # optional
        },
}
```

There are some additional options you can provide:

lang default interface lang (refer to PayU manual); default: `None`

signing for security reasons PayU can check a signature of all data that is sent from your service while redirecting to payment gateway; unless you really know what you are doing, this should be always on; default is True;

method the HTTP method used to connect with broker system on new payment; default is 'GET';

testing when you test your service you can enable this option, all payments for PayU will have a predefined "Test Payment" method which is provided by PayU service (needs to be enabled); default is False;

getpaid_configuration management command

After setting up django application it is also important to remember that some minimal configuration is needed also at PayU service configuration site. Please navigate to POS configuration, where you need to provide three links: success URL, failure URL, and online URL. The first two are used to redirect client after successful/failure payment. The third one is the address of script that will be notified about payment status change.

`getpaid.backends.payu` comes with `getpaid_configuration` management script that simplifies getting those links in your particular django environment. This is because you can customize path prefix when including urls from `getpaid`.

It will produce the following example output:

```
$. /manage.py  payu_configuration
Login to PayU configuration page and setup following links:

 * Success URL: http://example.com/getpaid.backends.payu/success/%orderId%/
               https://example.com/getpaid.backends.payu/success/%orderId%/

 * Failure URL: http://example.com/getpaid.backends.payu/failure/%orderId%/
               https://example.com/getpaid.backends.payu/failure/%orderId%/

 * Online  URL: http://example.com/getpaid.backends.payu/online/
               https://example.com/getpaid.backends.payu/online/

To change the domain name please edit Sites settings. Don't forget to setup your web␣
↪server to accept https connection in order to use secure links.

Request signing is ON
 * Please be sure that you enabled signing payments in PayU configuration page.
```

Warning: Please remember to set correct domain name in Sites framework.

Running celery for asynchronus tasks

This backend is asynchronous (as PayU requires an asynchronous architecture - they send a "ping" message that a payment change a status, and you need to asynchronously request theirs service for details of what has changed). That means that this backend requires django-celery application. Please refer to django-celery documentation for any additional information.

If you just want to make a quick start with using django-getpaid and django-celery please remember that after successful installation and enabling django-celery in your project you will need to run celery workers in order to process asynchronous task that this application requires. You can do that for example this way:

```
$ python manage.py celery worker --loglevel=info
```

1.4.3 Transferuj.pl backend `getpaid.backends.transferuj`

This backend can handle payment processing via Polish money broker Transferuj.pl.

Transferuj.pl accepts only payments in `PLN`.

Enable backend

Add backend full path to `GETPAID_BACKENDS` setting:

```
GETPAID_BACKENDS += ('getpaid.backends.transferuj', )
```

Don't forget to add backend path also to `INSTALLED_APPS`:

```
INSTALLED_APPS += ('getpaid.backends.transferuj', )
```

There is no need to add any urls definitions to main `urls.py` file, as they will be loaded automatically by `getpaid` application.

Setup backend

In order to start working with Transferuj.pl you will need to have an activated account in Transferuj.pl service. The following setup information is needed to be provided in the `GETPAID_BACKENDS_SETTINGS` configuration dict:

id Transferuj.pl client identificator,

key random (max. 16 characters long) string, that will be used for request security signing,

You need to provide this information in `GETPAID_BACKENDS_SETTINGS` dictionary:

```
GETPAID_BACKENDS_SETTINGS = {
    'getpaid.backends.transferuj' : {
            'id' : 123456,
            'key' : 'xxxxxxxxxxxxx',
            'signing' : True,       # optional
        },
}
```

There are some additional options you can provide:

signing for security reasons Transferuj.pl can check a signature of some data that is sent from your service while redirecting to payment gateway; unless you really know what you are doing, this should be always on; default is True;

method the HTTP method how to connect with broker system on new payment; default is 'GET';

allowed_ip Transferuj.pl requires to check IP address when they send you a payment status change HTTP request. By default, this module comes with list of hardcoded IP of Transferuj.pl system (according to the documentation). If you really need to you can override this list of allowed IP, setting this variable.

Note: Setting an empty list `[]` completely disables checking of IP address which is **NOT recommended**.

---

force_ssl_online default: False; this option when turned to True, will force getpaid to return an HTTPS URL for Transferuj.pl to send you payment status change.

> **Warning:** Remember to set Sites framework domain in database, as this module uses this address to build fully qualified URL.

force_ssl_return default: False; similarly to `force_ssl_online` but forces HTTPS for client returning links.

> **Warning:** Remember to set Sites framework domain in database, as this module uses this address to build fully qualified URL.

lang default interface lang; default: `None`

> **Warning:** It seems that this feature is undocumented. Transferuj.pl accepts `jezyk` parameter and I have this information from support (not from docs).

transferuj_configuration management command

After setting up django application it is also important to remember that some minimal configuration is needed also at Transferuj.pl service configuration site.

`getpaid.backends.transferuj` comes with `transferuj_configuration` management script that simplifies getting those links in your particular django eviroment. This is because you can customize path prefix when including urls from `getpaid`.

It will produce following example output:

```
$. /manage.py  transferuj_configuration
Please setup in Transferuj.pl user defined key (for security signing): xxxxxxxxxxxxx
```

> **Warning:** Please remember to set correct domain name in Sites framework.

1.4.4 Dotpay.eu backend `getpaid.backends.dotpay`

This backend can handle payment processing via Polish money broker Dotpay.pl/Dotpay.eu.

Dotpay.eu accepts payments in `PLN`, `EUR`, `USD`, `GBP`, `JPY`, `CZK`, `SEK`.

Setup backend

In order to start working with Dotpay you will need to have an activated account in Dotpay service.

Required keys:

id client ID

PIN secret used for checking messeges md5; generated in Dotpay admin panel

You need to provide this information in `GETPAID_BACKENDS_SETTINGS` dictionary:

```
GETPAID_BACKENDS_SETTINGS = {
    'getpaid.backends.dotpay' : {
            'id' : 123456,
            'PIN': 123456789,

        },
}
```

Optional keys:

force_ssl forcing HTTPS on incoming connections from Dotpay; default `False`

> **Warning: Set Sites domain name**
>
> This module requires Sites framework to be enabled. All backends base on Sites domain configuration (to generate fully qualified URL for payment broker service). Please be sure that you set a correct domain for your deployment before running `getpaid`.

method the HTTP method how to connect with broker system on new payment; default is 'GET';

lang default interface lang (refer to Dotpay manual); default: `None`

onlinetransfer if broker should show only online payment methods (refer to Dotpay manual); default: `False`

p_email custom merchant e-mail (refer to Dotpay manual); default: `None`

p_info custom merchant name (refer to Dotpay manual); default: `None`

tax 1% charity (refer to Dotpay manual); default: `False`

gateway_url You may want to change this to use dotpay testing account; default: `https://ssl.dotpay.eu/`

> **Warning:** Dotpay has strange policy regarding to gateway urls. It appears, that new accounts use `https://ssl.dotpay.pl/`, but old accounts, that have 5-digit long id's, should still use old gateway `https://ssl.dotpay.eu/`. For reasons of this behaviour You need to contact dotpay support.

1.4.5 Przelewy24 backend `getpaid.backends.przelewy24`

This backend can handle payment processing via Polish money broker Przelewy24.pl.

Przelewy24 accepts payments in `PLN`.

Acknowledgements: Przelewy24 backend was kindly funded by Issue Stand.

Setup backend

In order to start working with Przelewy24 you will need to have an activated account in Przelewy24 service.

Required keys:

id client ID

crc CRC code for client ID

You need to provide this information in `GETPAID_BACKENDS_SETTINGS` dictionary:

```
GETPAID_BACKENDS_SETTINGS = {
    'getpaid.backends.przelewy24' : {
            'id' : 123456,
            'crc': 'fc1c0644f644fcc',
        },
}
```

Optional keys:

sandbox set `True` to use sandbox environment; default `False`

lang default interface lang if not overridden by signal (`'pl', 'en', 'es', 'de', 'it'`); default: `None`

ssl_return set this option to `True` if a client should return to your site after payment using HTTP SSL; default `False`

Credit Card payments

To enable credit card payments you may want to provide some additional required information to getpaid via signal query:

```
def user_data_query_listener(sender, order=None, user_data=None, **kwargs):
    """
    Here we fill some static user data, just for test
    """
    user_data['p24_klient'] = u'Jan Nowak'
    user_data['p24_adres'] = u'ul. Ulica 11'
    user_data['p24_kod'] = u'00-000'
    user_data['p24_miasto'] = u'Warszawa'
    user_data['p24_kraj'] = u'PL'

signals.user_data_query.connect(user_data_query_listener)
```

Additional info

Przelewy24 naively assumes that all payments will end up with successful client redirection to the source webpage. According to documentation this redirection is also a signal for checking payment status. However, as you can easily imagine, client could close browser before redirection from payment site. Przelewy24 suggest that you could deliver them via e-mail additional URL that will be requested in such case (after 15 min delay).

This is strongly recommended (as you may never receive some transactions confirmations), you can generate appropriate URL (to your django installation) using management command:

```
$ python manage.py przelewy24_configuration
Please contact with Przelewy24 (serwis@przelewy24.pl) and provide them with the␣
↪following URL:

http://mydomain.com/getpaid.backends.przelewy24/online/

This is an additional URL for accepting payment status updates.

To change domain name please edit Sites settings. Don't forget to setup your web␣
↪server to accept https connection in order to use secure links.

Sandbox mode is ON.
```

1.4.6 Moip backend `getpaid.backends.moip`

This backend can handle payment processing via brazilian money broker Moip.com.br.

Moip accepts payments exclusively in `BRL`.

Setup backend

In order to start working with Moip you will need to have an activated account with Moip.

Required keys:

token your seller's account token

key your secret key

Optional keys:

testing if set to true it will use sandox' URL. Default value is false

You need to provide this information in `GETPAID_BACKENDS_SETTINGS` dictionary:

```
GETPAID_BACKENDS_SETTINGS = {
    'getpaid.backends.moip' : {
            'key': 'AB310XDOPQO13LXPAO',
            'token': "AB310XDOPQO13LXPAO",
            'testing': True,
        },
}
```

Status changes

Even though Moip has 9 different statuses, this only translates into 3 statuses in *django-getpaid*. Before the payment is made, the initial status is *in_progress*. Once it moves in Moip to the authorized, the getpaid state also changes on this backend to paid. If at any point Moip changes the transaction status to chargeback or refunded, the status on this backend will also enter the failed state. Beware that all others statuses in between are ignored. You will not be notified if a transaction moves from paid to available or if it enters dispute. This should however make no difference, as it only really matters if your transaction at Moip changes from in dispute to refunded or chargedback (and both are tracked).

1.4.7 Paymill backend `getpaid.backends.paymill`

This backend can handle payment processing via the "Stripe for Europe" Paymill.

Paymill accepts payments in `EUR`, `CZK`, `DKK`, `HUF`, `ISK`, `ILS`, `LVL`, `CHF`, `NOK`, `PLN`, `SEK`, `TRY` and `GBP`.

Setup backend

In order to start working with Paymill you will need to have an activated account with Paymill.

Required keys:

PAYMILL_PUBLIC_KEY your public key

PAYMILL_PRIVATE_KEY your private key

You need to provide this information in `GETPAID_BACKENDS_SETTINGS` dictionary:

```
GETPAID_BACKENDS_SETTINGS = {
    'getpaid.backends.paymill': {
        'PAYMILL_PUBLIC_KEY': '024436912481f223e137769e2886830b',
        'PAYMILL_PRIVATE_KEY': '1b9a36f6g6e2d52aab7858f5a5eb8k67',
    }
}
```

A word about security

Though we have to display the form for the credit card data on our website, it will never be sent to the server to comply with the Payment Card Industry Data Security Standard. Instead, Paymill's JavaScript API is used to generate a token that is sent to the server to process the payment.

Integration into your website

You can (and should) overwrite the `getpaid_paymill_backend/paymill.html` file, but be sure to both include the form as well as the `getpaid_paymill_backend/paymill_form.html` file that shows the actual form and handles the JavaScript.

1.5 Writing custom payment backend

django-getpaid allows you to use two types of custom backends: internal and third-party backends. There is no architectural difference between them, only the first one is shipped with django-getpaid code, while the second one can be maintained separately. However if you are going to provide a payment backend to any popular payment method (even if it's only popular in your country) you are very welcome to contribute your backend to django-getpaid. Some hints how to do that are described in *Welcome to django-getpaid's documentation!* "Developing" section.

1.5.1 Creating initial backend application

All payment backends are standalone django apps. You can easily create one using `django-admin.py` tool:

```
$ django-admin.py startapp mybackend
```

Before using this app as a legal django-getpaid backend, you need to follow a few more steps described below.

1.5.2 Creating `PaymentProcessor` class

Required

`PaymentProcessor` class is one of the most important parts of whole backend. All logic related to processing a payment should be put into this class. This class should derive from `getpaid.backends.PaymentProcessorBase`.

class getpaid.backends.**PaymentProcessorBase**(*payment*)

Base for all payment processors. It should at least be able to:

- redirect to a gateway based on Payment object
- manage all necessary logic to accept payment from gateway, e.g. expose a View for incoming transaction notification status changes

BACKEND = None

BACKEND_ACCEPTED_CURRENCY = ()

BACKEND_LOGO_URL = None

BACKEND_NAME = None

classmethod get_backend_setting(*name*, *default=None*)
Reads `name` setting from backend settings dictionary.

If *default* value is omitted, raises `ImproperlyConfigured` when setting `name` is not available.

get_form(*post_data*)
Only used if the payment processor requires POST requests. Generates a form only containg hidden input fields.

get_gateway_url(*request*)
Should return a tuple with the first item being the URL that redirects to payment Gateway Second item should be if the request is made via GET or POST. Third item are the parameters to be passed in case of a POST REQUEST. Request context need to be given because various payment engines requires information about client (e.g. a client IP).

classmethod get_logo_url()
Get backend logo. Use always this method, instead of reading BACKEND_LOGO_URL attribute directly.

Returns str

get_order_description(*payment*, *order*)
Renders order description using django template provided in `settings.GETPAID_ORDER_DESCRIPTION` or if not provided return unicode representation of `Order` object.

Your `PaymentProcessor` needs to be named exactly this way and can live anywhere in the code structure as long as it can be imported from the main scope. We recommend you to put this class directly into your app `__init__.py` file, as there is really no need to complicate it anymore by adding additional files.

1.5.3 Overriding `get_gateway_url()` method

Required

This is the most important method from the django-getpaid perspective. You need to override the `get_gateway_url` method, which is an entry point to your backend. This method is based on the `request` context and on the `self.payment` and should return the URL to the payment gateway that the client will be redirected to.

If your backend emits the `getpaid.signals.user_data_query` signal, please respect the convention below on which key names to expect as parameters. The objective is to make this signal as agnostic as possible to payment processors.

- email
- lang
- name
- address
- address_number
- address_complement
- address_quarter
- address_city
- address_state
- address_zip_code
- phone
- phone_area_code

1.5.4 Providing extra models

Required (providing `build_models()` function)

Your application in most cases will not need to provide any models at all. In this situation it is very important to add following line in your `models.py` empty file:

```
def build_models(payment_class):
    return []
```

The method `build_models` is required for every payment backend as it allows to dynamically build django models in run-time, that need to depend on `Payment` class but don't want to use `content_type` in django.

To do that you will need to use `getpaid.abstract_mixin.AbstractMixin`, please refer to code. Here is just a simple example of a working dynamically created model to give you an idea of how it works:

```
from django.db import models
from getpaid.abstract_mixin import AbstractMixin

class PaymentCommentFactory(models.Model, AbstractMixin):
    comment = models.CharField(max_length=100, default="a dummy transaction")

    class Meta:
        abstract = True

    @classmethod
    def contribute(cls, payment):
        return {'payment': models.OneToOneField(payment)}

PaymentComment = None

def build_models(payment_class):
    global PaymentComment
```

(continues on next page)

(continued from previous page)

```
    class PaymentComment(PaymentCommentFactory.construct(payment_class)):
        pass
    return [PaymentComment]
```

This will create in run-time a model `PaymentComment` which has two fields: CharField `comment` and ForeignKey `payment`. You can use it in your backend.

Note: Obviously you can also provide static django models without using this fancy method, as all backend apps are also regular django apps!

1.5.5 Providing extra urls

Optional

In most cases your backend will need to provide some additional urls - for example the url for accepting incoming request from payment gateways about status changes, etc. You can just add your URL definitions like in standard django app.

Note: You don't need to register your backend's `urls.py` module with `include()` in your original project's `urls.py`. All enabled applications will have theirs urls automatically appended, but they will be prefixed with backend full path.

For example, consider following case:

```
from django.conf.urls import patterns, url
from getpaid.backends.dummy.views import DummyAuthorizationView

urlpatterns = patterns('',
    url(r'^payment/authorization/(?P<pk>[0-9]+)/$', DummyAuthorizationView.as_view(),␣
↪name='getpaid:dummy:authorization'),
)
```

This will expose a link that will point to something like: `/getpaid.backends.dummy/payment/authorization/0/` (of course `getpaid.urls` could be prefixed with some other path, then the whole path would also have some additional prefix e.g. `/my/app/payments/getpaid.backends.dummy/payment/authorization/0/`). As you can see like in regular django app, you connect your urls with app views.

1.5.6 Providing extra views

Optional

If you need to provide some views, please use standard `views.py` django convention. Class based views are welcome!

Note: It is highly recommended to manage all payment logic in additional methods of `PaymentProcessor` class. Let the view only be a wrapper for preparing arguments for one `PaymentProcessor` logic method. In this way you will keep all payment processing related logic in one place.

Warning: When using any kind of POST views that accepts external connections remember to use `@csrf_exempt` decorator, as django by default will 403 Forbid those connections.

1.5.7 Asynchronous tasks

Optional

django-getpaid is highly recommending using django-celery for all asynchronous processing. If you need to do any, please create celery tasks with the `@task()` decorator.

Note: Just like what we have done with the view, when processing celery tasks it is recommended to put your business logic in the class `PaymentProcessor`. Let the task function only be a wrapper for preparing arguments for one `PaymentProcessor` logic method.

1.5.8 Configuration management script

Optional

If your module need to generate any kind of configuration data (for example links that you should provide in payment broker configuration site) you should create a django management script that displays all needed information (e.g. displaying links using django `reverse()` function). By the convention you should name this management script in format: `<short backend name>_configuration.py` (e.g. `payu_configuration.py`).

1.6 South migrations

django-getpaid does not provide south migrations by itself, because it's models depend on your main project models. It means that some of getpaid models are generated on the fly and unique through each application. Therefore creating one common set of migrations is not possible.

However database migrations can be easily managed using a great feature of South module, accesible via SOUTH_MIGRATION_MODULES setting.

This option allows you to overwrite default South migrations search path and create your own project dependent migrations in scope of your own project files. To setup custom migrations for your project follow these simple steps.

1.6.1 Step 1. Add `SOUTH_MIGRATION_MODULES` setting

You should put your custom migrations somewhere. The good place seems to be path `PROJECT_ROOT/migrations/getpaid` directory.

Note: Remember that `PROJECT_ROOT/migrations/getpaid` path should be a python module, i.e. it needs to be importable from python.

Then put the following into `settings.py`:

```
SOUTH_MIGRATION_MODULES = {
    'getpaid' : 'yourproject.migrations.getpaid',
}
```

you can also track migrations for particular backends:

```
SOUTH_MIGRATION_MODULES = {
    'getpaid' : 'yourproject.migrations.getpaid',
    'payu' : 'yourproject.migrations.getpaid_payu',
}
```

1.6.2 Step 2. Create initial migration

From now on, everything works like standard South migrations, with the only difference that migrations are kept in scope of your project files - not getpaid module files.

```
$ python migrate.py schemamigration --initial getpaid
```

1.6.3 Step 3. Migrate changes on deploy

Make sure to run migrate for your app containing your order model before running the getpaid migrate.

```
$ python migrate.py schemamigration --initial orders
$ python migrate.py migrate orders
$ python migrate.py migrate getpaid
```

1.6.4 Step 4. Upgrading to new a version of getpaid

When there is a new version of getpaid, you can upgrade your module by simply using South to generate custom migration:

```
$ python migrate schemamigration --auto getpaid
```

and then:

```
$ python migrate.py migrate getpaid
```

CHAPTER 2

Versioning

Semantic Version guidelines will be followed in this project versioning.

Releases will be numbered with the following format:

`<major>.<minor>.<patch>`

And constructed with the following guidelines:

- Breaking backward compatibility bumps the major (and resets the minor and patch)
- New additions without breaking backward compatibility bumps the minor (and resets the patch)
- Bug fixes and misc changes bumps the patch
- Major version `0` means early development stage

For more information on SemVer, please visit http://semver.org/.

CHAPTER 3

Developing

Project leader:

- Krzysztof Dorosz <cypreess@gmail.com>.

Contributors:

- Dominik Kozaczko <http://dominik.kozaczko.info>
- Bernardo Pires Carneiro <https://github.com/bcarneiro>

You are welcome to contribute to this project via github fork & pull request.

If you create your standalone backend (even if you don't want to incorporate it into this app) please write to me, you can still be mentioned as third party payment backend provider.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

B

G

P

R