plata Documentation

Release v1.1.0-497-g46d14d9

Feinheit AG

Oct 11, 2018

Contents

This is the documentation for fiee's fork of Feinheit's project. Find the sources at https://github.com/fiee/plata

CHAPTER 1

Overview

Instead of fighting against a big and all-knowing shop system, which never does the right thing although it "only requires pushing a few configuration buttons here and there", Plata caters to the programmer by only providing order, payment and discount models and some code for cart, checkout and payment views. Building a shop on top of Django and Plata still requires programming – but that's what we do after all.

CHAPTER 2

Contents

2.1 Installation instructions

2.1.1 Installation

This document describes the steps needed to get Plata up and running.

Plata is based on Django, so you need a working Django installation first. Plata is developed using Django 1.10, and is not tested against any earlier version.

You can download a version of plata using `pip`:

```
$ pip install -e git+https://github.com/fiee/plata@next#egg=Plata
```

Please note that the package installable with `pip` only contains the files needed to run plata. It does not include documentation, tests or the example project which comes with the development version, which you can download using the Git version control system:

```
$ git clone git://github.com/fiee/plata.git -b next
```

Plata requires a version of simplejson >=2.1 because older versions cannot serialize decimal values, only floats.

In addition, you will need PDFDocument if you want to generate PDFs.

2.1.2 Configuration

Writing your product model

First, you need to write your own product and price model. There is a small (and documented) interface contract described in *Contracts*.

The smallest possible implementation while still following best practice follows here:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

from plata.product.models import ProductBase
from plata.shop.models import PriceBase

class Product(ProductBase):
    name = models.CharField(_('name'), max_length=100)
    slug = models.SlugField(_('slug'), unique=True)
    description = models.TextField(_('description'), blank=True)

    class Meta:
        ordering = ['name']
        verbose_name = _('product')
        verbose_name_plural = _('products')

    def __unicode__(self):
        return self.name

    @models.permalink
    def get_absolute_url(self):
        return ('plata_product_detail', (), {'slug': self.slug})

class ProductPrice(PriceBase):
    product = models.ForeignKey(Product, verbose_name=_('product'),
        related_name='prices')

    class Meta:
        get_latest_by = 'id'
        ordering = ['-id']
        verbose_name = _('price')
        verbose_name_plural = _('prices')
```

Plata has to know the location of your shop model, because it is referenced e.g. in the `product` ForeignKey of order items. If the product model exists in the `myapp` Django application, add the following setting:

```
PLATA_SHOP_PRODUCT = 'myapp.Product'
```

Adding the modules to `INSTALLED_APPS`

```
INSTALLED_APPS = (
    ...
    'myapp',
    'plata',
    'plata.contact', # Not strictly required (contact model can be exchanged)
    'plata.discount',
    'plata.payment',
    'plata.shop',
    ...
    )
```

You should run `./manage.py makemigrations plata` and `./manage.py migrate plata` after you've added the required modules to `INSTALLED_APPS`.

Creating the `Shop` object

Most of the shop logic is contained inside `Shop`. This class implements cart handling, the checkout process and handing control to the payment modules when the order has been confirmed. There should exist exactly one Shop instance in your site (for now).

The Shop class requires three models and makes certain assumptions about them. The aim is to reduce the set of assumptions made or at least make them either configurable or overridable.

The models which need to be passed when instantiating the Shop object are

- `Contact`
- `Order`
- `Discount`

Example:

```
from plata.contact.models import Contact
from plata.discount.models import Discount
from plata.shop.models import Order
from plata.shop.views import Shop

shop = Shop(
    contact_model=Contact,
    order_model=Order,
    discount_model=Discount,
    )
```

The shop objects registers itself in a central place, and can be fetched from anywhere using:

```
import plata
shop = plata.shop_instance()
```

The `Shop` class instantiation may be in `myapp.urls` or `myapp.views` or somewhere similar, it's recommended to put the statement into the `views.py` file because the `Shop` class mainly offers views (besides a few helper functions.)

Adding views and configuring URLs

The `Shop` class itself does not define any product views. You have to do this yourself. You may use Django's generic views or anything else fitting your needs.

Generic views using `plata.shop_instance()` could look like this:

```
from django import forms
from django.contrib import messages
from django.shortcuts import get_object_or_404, redirect, render
from django.utils.translation import ugettext_lazy as _
from django.views import generic

import plata
from plata.contact.models import Contact
from plata.discount.models import Discount
from plata.shop.models import Order
from plata.shop.views import Shop

from myapp.models import Product
```

(continues on next page)

(continued from previous page)

```
shop = Shop(
    contact_model=Contact,
    order_model=Order,
    discount_model=Discount,
    )

product_list = generic.ListView.as_view(
    queryset=Product.objects.all(),
    paginate_by=10,
    template_name='product/product_list.html',
    )


class OrderItemForm(forms.Form):
    quantity = forms.IntegerField(label=_('quantity'), initial=1,
        min_value=1, max_value=100)


def product_detail(request, slug):
    product = get_object_or_404(Product, slug=slug)

    if request.method == 'POST':
        form = OrderItemForm(request.POST)

        if form.is_valid():
            # Referencing the shop object instantiated above
            order = shop.order_from_request(request, create=True)
            try:
                order.modify_item(product, relative=form.cleaned_data.get('quantity'))
                messages.success(request, _('The cart has been updated.'))
            except forms.ValidationError, e:
                if e.code == 'order_sealed':
                    [messages.error(request, msg) for msg in e.messages]
                else:
                    raise

            return redirect('plata_shop_cart')
    else:
        form = OrderItemForm()

    return render(request, 'product/product_detail.html', {
        'object': product,
        'form': form,
        })
```

Next, you need to add the Shop's URLs to your URLconf:

```
from django.conf.urls import include, url
from myapp.views import shop, product_list, product_detail

urlpatterns = [
    url(r'^shop/', include(shop.urls)),
    url(r'^products/$',
        product_list,
        name='plata_product_list'),
    url(r'^products/(?P<slug>[-\w]+)/$',
        product_detail,
```

(continues on next page)

(continued from previous page)

```
        name='plata_product_detail'),
]
```

The context processor

You can add `plata.context_processors.plata_context` to your `settings.TEMPLATE_CONTEXT_PROCESSORS`. This will add the following variables to your template context if they are available:

- `plata.shop`: The current `plata.shop.views.Shop` instance
- `plata.order`: The current order
- `plata.contact`: The current contact instance
- `plata.price_includes_tax`: Whether prices include tax or not

Alternatively you can also just overwrite the `get_context` method in your shop class.

Setting up logging

Plata uses Python's logging module for payment processing, warnings and otherwise potentially interesting status changes. The logging module is very versatile and sometimes difficult to configure, because of this an example configuration is provided here. Put the following lines into your `settings.py`, adjusting the logfile path:

```
import logging, os
import logging.handlers

LOG_FILENAME = os.path.join(APP_BASEDIR, 'log', 'plata.log')

plata_logger = logging.getLogger('plata')
plata_logger.setLevel(logging.DEBUG)
plata_logging_handler = logging.handlers.RotatingFileHandler(LOG_FILENAME,
    maxBytes=10*1024*1024, backupCount=15)
plata_logging_formatter = logging.Formatter('%(asctime)s %(levelname)s:%(name)s:
↪%(message)s')
plata_logging_handler.setFormatter(plata_logging_formatter)
plata_logger.addHandler(plata_logging_handler)
```

Implementing the shop as FeinCMS application content

To use the shop as application content you have to overwrite the `render` and `redirect` methods on your shop class. Take a look at this example in `myshop.views`:

```
from feincms.content.application.models import app_reverse
from plata.shop.views import Shop

class CustomShop(Shop):

    def render(self, request, template, context):
        """ render for application content """
        return template, context

    def redirect(self, url_name):
```

(continues on next page)

(continued from previous page)

```
        return redirect(app_reverse(url_name, 'myshop.urls'))

    base_template = 'site_base.html'

shop = CustomShop(Contact, Order, Discount)
```

2.2 Payment support

Plata supports the following means of payment:

- *Cash on delivery*
- *Payment in advance*
- *Check / Bank Transfer*
- *Paypal*
- *Postfinance (Switzerland)*
- *Ogone*
- *Datatrans (Switzerland)*
- *Billogram (Sweden)*
- *Payson (Sweden)*
- *PagSeguro (Brazil)*
- *Stripe Checkout*

''Beware'', most of these are not up to date and their general state of usefulness is unknown. Always try to understand the source code.

2.2.1 Cash on delivery

This payment module does not need any configuration. It simply creates an order payment with the order total as amount and confirms the order.

2.2.2 Payment in advance

Send an invoice to the customer and wait for clearance by an admin. This is just a copy of the check module without mentioning checques.

Configuration: `PLATA_PAYMENT_PREPAY_NOTIFICATIONS`

2.2.3 Check / Bank Transfer

Configuration: `PLATA_PAYMENT_CHECK_NOTIFICATIONS`

2.2.4 Paypal

The Paypal payment module needs two configuration values:

```
PAYPAL = {
    'LIVE': False, # Use sandbox or live payment interface?
    'BUSINESS': 'paypal@example.com',
    'IPN_SCHEME': 'https',
    'RETURN_SCHEME': 'https'
}
```

The default IPN URL is `payment/paypal/ipn/`; the base path is defined by where you added `shop.urls`.

You should also specify *'RETURN_SCHEME'* and *'IPN_SCHEME'* within the *PAYPAL* settings and use SSL; *IPN_SCHEME* defaults to *'http'*, and *RETURN_SCHEME* will auto-select *'http'* or *'https'* based on the *request.is_secure()* value of the payment initiating request.

Never use unencrypted communication in production!

2.2.5 Stripe Checkout

Latest addition to Plata's payment modules, not yet tested in production.

Configuration values:

```
STRIPE = {
    'PUBLIC_KEY': '...',
    'SECRET_KEY': '<secret>',
    'LOGO': '%simg/yoursiteslogo.png' % STATIC_URL,
    'template': 'myshop/payment/stripe_form.html',
}
```

Testing or live payments depend on the keys; you might use different setup files. Don't keep secrets (passwords etc.) in setting files that get checked in into source control!

If you use zero-decimal currencies like *'JPY'* or *'KRW'*, you should setup:

```
CURRENCIES_WITHOUT_CENTS = ('JPY', 'KRW')  # these two are default
```

And this is *stripe_form.html*:

```
{% if not callback %}
{% trans "Thank you for your payment!" %}
{% include "plata/_order_overview.html" %}

<form action="{{ post_url }}" method="POST">
{% csrf_token %}
{{ form.management_form }}
{{ form.id }}
  <script
    src="https://checkout.stripe.com/checkout.js" class="stripe-button"
    data-label="{% trans 'Pay with Card' %}"
    data-key="{{ public_key }}"
    data-amount="{{ amount }}"
    data-currency="{{ currency }}"
    data-name="{{ name }}"
    data-description="{{ description }}"
```

(continues on next page)

(continued from previous page)

```
    data-image="{{ logo }}"
    data-locale="auto"
    {% if user.email %}data-email="{{ user.email }}"{% endif %}
    data-zip-code="true">
  </script>
</form>
{% endif %}
```

More information: https://stripe.com/docs/checkout

2.2.6 Postfinance (Switzerland)

The Postfinance payment module requires the following configuration values:

```
POSTFINANCE = {
    'LIVE': False,
    'PSPID': 'exampleShopID',
    'SHA1_IN': '<shared secret>',
    'SHA1_OUT': '<shared secret>',
    }
```

This module implements payments using SHA-1 hash validation with the simpler SHA-1 IN hashing algorithm. (Someone should update this to a safer method!)

The default server to server notification URL is `payment/postfinance/ipn/`; the base path is defined by where you added `shop.urls`.

More information:

- https://www.postfinance.ch/de/unternehmen/produkte/debitorenloesungen/e-payment-zahlungsarten.html
- https://www.postfinance.ch/de/unternehmen/produkte/debitorenloesungen/e-payment-psp.html

2.2.7 Datatrans (Switzerland)

The Datatrans payment module requires the following configuration values:

```
DATATRANS = {
    'MERCHANT_ID': '1000000000000',
    'LIVE': False
    }
```

To work with them, you must inquire about contracts.

In their lowest pay scale they charge about 500 CHF for setup plus 20 CHF per month plus transaction fees (in 2017). That was too much for my customer, so I stopped updating the datatrans payment module.

2.2.8 Billogram (Sweden)

Requires `billogram.api` and the following configuration:

```
BILLOGRAM = {
    'API_USER': '',
    'API_AUTHKEY': '',
```

(continues on next page)

(continued from previous page)

```
    'API_BASE': '',
    'SIGN_KEY': ''
}
```

2.2.9 Payson (Sweden)

Requires `payson_api` and the following configuration values:

```
PAYSON = {
    'USER_ID': '',
    'USER_KEY': '',
    'EMAIL': 'you@example.com'
    }
```

2.2.10 PagSeguro (Brazil)

The PagSeguro payment module is looking for the following configuration values:

```
PAGSEGURO = {
    'EMAIL': 'you@example.com',
    'TOKEN': '???',
    'LOG': 'pagseguro.log'  # file name
    }
```

2.2.11 Ogone

The Ogone payment module requires the following configuration values:

```
OGONE = {
    'LIVE': False,
    'PSPID': 'exampleShopID',
    'SHA1_IN': '<shared secret>',
    'SHA1_OUT': '<shared secret>',
    }
```

This module implements payments using SHA-1 hash validation with the simpler SHA-1 IN hashing algorithm.

The default server to server notification URL is `payment/ogone/ipn/`; the base path is defined by where you added `shop.urls`.

This payment provider is part of Ingenico since 2014, thus it's questionable if the module still works.

2.3 Settings

PLATA_PRICE_INCLUDES_TAX: Determines whether prices are shown with tax included by default. This setting does not influence internal calculations in any way.

Defaults to `True`

PLATA_ORDER_PROCESSORS: The list of order processors which are used to calculate line totals, taxes, shipping cost and order totals.

The classes can be added directly or as a dotted python path. All classes should extend `ProcessorBase`.

PLATA_PAYMENT_MODULES: The list of payment modules which can be used to pay the order. Currently, all available modules are enabled too.

PLATA_PAYMENT_MODULE_NAMES The user-visible names of payment modules can be modified here. Example:

```
PLATA_PAYMENT_MODULE_NAMES = {'paypal': _('Paypal and credit cards')}
```

PLATA_SHIPPING_FIXEDAMOUNT: If you use `FixedAmountShippingProcessor`, you should fill in the cost incl. tax and tax rate here.

Defaults to `{'cost': Decimal('8.00'), 'tax': Decimal('7.6')}`

PLATA_ZIP_CODE_LABEL: Since ZIP code is far from universal, and more an L10N than I18N issue... Defaults to `'ZIP code'`.

PLATA_SHIPPING_WEIGHT_UNIT and PLATA_SHIPPING_LENGTH_UNIT: If you use `Postage` and don't like metric units, you can change them here. Defaults to `'g'` resp. `'mm'`. No calculations involved, just a display string.

PLATA_REPORTING_STATIONERY: Stationery used by PDFDocument to render invoice and packing slip PDFs.

PLATA_PDF_FONT_NAME: Custom regular font name to be used by PDFDocument for rendering PDF invoices. Defaults to `''` (using default of `reportlab`).

PLATA_PDF_FONT_PATH Custom regular font path to be used by PDFDocument for rendering PDF invoices. Defaults to `''`.

PLATA_PDF_FONT_BOLD_NAME Custom bold font path to be used by PDFDocument for rendering PDF invoices. Defaults to `''`.

PLATA_PDF_FONT_BOLD_PATH Custom bold font path to be used by PDFDocument for rendering PDF invoices. Defaults to `''`.

PLATA_STOCK_TRACKING: Accurate transactional stock tracking. Needs the `plata.product.stock` Django application.

Each stock change will be recorded as a distinct entry in the database. Products will be locked when an order is confirmed for 15 minutes, which means that it's not possible to begin or end the checkout process when stock is limited and someone else has already started paying.

CURRENCIES: A list of available currencies. Defaults to `('CHF', 'EUR', 'USD', 'CAD')`. You should set this variable for your shop.

PLATA_SHOP_PRODUCT: Target of order item product foreign key in `app_label.model_name` notation. Defaults to `'product.Product'`

2.4 Contracts

This document describes the minimum contract the shop models are supposed to fulfill.

The simple example at `examples/simple/` is supposed to demonstrate the most simple use of Plata.

2.4.1 Product model

The (base) product model has to be specified as `PLATA_SHOP_PRODUCT` using Django's `app_label.model_name` notation. The model referenced will be the model pointed to by the `OrderItem` order line items.

Since it's your responsibility to write all views and URLs for your product catalogue, there aren't many things Plata's products have to fulfill.

Plata provides an abstract model at `plata.product.models.ProductBase`. You do not have to use it - it just provides the standard interface to the price determination routines already.

- `get_price(currency=None, orderitem=None):`

  The `get_price` method has to accept at least the current currency and optionally more arguments too, such as the current line item for the implementation of price tiers.

  This method must return a `Price` instance. It has to raise a `Price.DoesNotExist` exception if no price could be found for the passed arguments.

- `handle_order_item(self, orderitem):`

  This method is called when modifying order items. This method is responsible for filling in the `name` and `sku` (Stock Keeping Unit) field on the order item.

Different product models

If you need different product models, consider using Django's model inheritance and something like django-polymorphic.

2.4.2 Price model

Plata has a bundled abstract price model which does almost everything required. You have to provide the concrete price model yourself though.

The only thing it lacks for a basic price is a foreign key to product. `ProductBase.get_price` assumes that the foreign key is defined with `related_name='prices'`.

When modifying a cart or an order, Plata will call the price object's `handle_order_item` method, passing the order item instance as the only argument. This method is responsible for filling the following attributes on the order item:

- `_unit_price`: Unit price excl. tax.
- `_unit_tax`: The tax on a single unit.
- `tax_rate`: The applied tax rate as a percentage. This is slightly redundant as it could be calculated from `_unit_price` and `_unit_tax`.
- `tax_class`: (Optional) a foreign key to `plata.shop.models.TaxClass`. This is purely informative. This field is not mandatory.
- `is_sale`: Boolean flag denoting whether the price is a sale price or not. This flag is unconditionally set to `False` by the price base class.

The price model offers the following attributes:

- `unit_price_excl_tax`

  A `decimal.Decimal` describing the unit price with tax excluded. `unit_price_incl_tax` and `unit_price` are offered by the default implementation too, but they aren't mandatory.

- `unit_tax`

  A `decimal.Decimal` too. This is the tax amount per unit, **not** the tax rate.

- `tax_class.rate`

  The tax rate as a percentage, meaning `19.6` for a tax rate of 19.6%.

- `tax_class`

  A `plata.shop.models.TaxClass` object for the given price.

2.4.3 Contact model

The example at `examples/custom` shows how the contact model might be replaced with a different model. The following fields and methods are mandatory:

- `Contact.user`: `ForeignKey` or `OneToOneField` to `auth.User`.
- `Contact.update_from_order(self, order, request=None)`: Updates the contact instance with data from the passed order (and from the optional request).
- `Contact.ADDRESS_FIELDS`: A list of fields which are available as billing and/or shipping address fields.

The last point isn't strictly necessary and can be circumvented by overriding `plata.shop.views.Shop.checkout_form()`.

`plata.contact.models.Contact` offers all that, and more.

2.5 Taxes

Plata supports different tax rates, even different tax rates in the same order. When calculating the order total, tax amounts with the same tax rate are grouped and can be shown separately on an invoice document.

The tax details are stored in the `data` attribute on the order instance. The format is as follows:

```
order.data['tax_details'] = [
    [<tax_rate>, {
        'discounts': <sum of all discounts>,
        'prices': <sum of line item prices>,
        'tax_amount': <sum of line item tax amounts>,
        'tax_rate': <tax rate (redundant)>,
        'total': <sum of line item totals>,
        }],
    # Another [tax_rate, {details}] instance etc.
    ]
```

The PDF code in `plata.reporting.order` demonstrates how the tax details might be used when generating an invoice.

2.6 Discounts

Plata comes with a discount implementation which already implements a wide range of possible discounting strategies. Several factors decide whether a certain discount is active:

- Its `is_active` flag.

- The validity period, determined by two dates, `valid_from` and `valid_until`.
- The number of allowed uses, `allowed_uses`. An usage example is limiting a discount to the first ten customers using it.

Plata knows three distinct discount types:

- An amount discount (given with tax included or excluded) which is applied to the subtotal. This type reduces the subtotal, the total and the tax amount.
- A percentage discount which is applied to the subtotal. Like the aforementioned amount type, this type reduces the subtotal, the total and the tax amount.
- A discount code usable as a means of payment. This discount type only leads to a reduced total and does not change the subtotal or the tax amount.

Discounts should not always be applicable to all products which can be bought. Because Plata does not come with a product model it also does not come with many strategies for selecting eligible products.

Bundled strategies are:

- `all`: All products are eligible.
- `exclude_sale`: Order items which have their `is_sale` flag set to `True` are excluded from discounting.

It's quite easy to add additional strategies. If you have a category model and only want to allow products from a set of categories for discounting, that's the code you need:

```
from django import forms
from django.db.models import Q
from django.utils.translation import ugettext_lazy as _
from plata.discount.models import DiscountBase
from myapp.models import Category

DiscountBase.CONFIG_OPTIONS.append(('only_categories', {
    'title': _('Only products from selected categories'),
    'form_fields': [
        ('categories', forms.ModelMultipleChoiceField(
            Category.objects.all(),
            label=_('categories'),
            required=True,
            )),
        ],
    'product_query': lambda categories: Q(categories__in=categories),
    }))
```

A different requirement might be that discounts can only be applied if a product will be bought several times (granted, that's a bit a stupid idea but it demonstrates how eligible products can be determined by their cart status):

```
from django import forms
from django.db.models import Q
from django.utils.translation import ugettext_lazy as _
from plata.discount.models import DiscountBase
from myapp.models import Category

DiscountBase.CONFIG_OPTIONS.append(('only_multiple', {
    'title': _('Only products which will be bought several times'),
    'orderitem_query': lambda **values: Q(quantity__gt=1),
    }))
```

Discounts are configured with the following variables:

- The key, given as first argument in the `(key, configuration)` tuple.
- `title`: A human-readable title.
- `form_fields`: A list of `(field_name, field_instance)` form fields which will be shown in the Django administration. Note: The fields are only visible after saving the discount once.
- `product_query`: The values from the form fields above will be passed as keyword arguments, the return value is passed to a `filter()` call on the product model queryset.
- `orderitem_query`: The values frmo the form fields above will be passed as keyword arguments, the return value is passed to a `filter()` call on the order items queryset.

`form_fields` is optional, one of `product_query` and `orderitem_query` should always be provided. This is not enforced by the code however.

2.7 Shipping

This (new and still incomplete) module will provide the calculation of shipping costs, depending on the size and weight of your products.

You can register the tariffs of several shipping providers (postal services) for groups of countries that cost the same postage and several package sizes.

Setup:

- Include *plata.shipping* in your *INSTALLED_APPS*.
- Run *manage.py makemigrations shipping* and *manage.py migrate*.
- Login to your admin interface and ...
- Create some country groups – the first will be your home country, others depend on the tariff groups of your shipping provider, like "European Union", "World 1", "World 2" etc.
- Define the countries that belong to these groups.
- Setup the shipping providers you work with, depending on the countries they serve.
- At last it's the most work to register all the different postage tariffs.

Some providers have maximum sizes for each dimension (length, weidth, height), others calculate by adding these. Some calculate by "tape measure", i.e. combined length and girth (in German: Gurtmaß), you must translate that to our "3D" sum yourself. *Postage.max_size* is the smaller of either the sum of length, width and height or *max_3d*.

Because also the packaging has a considerable weight that depends on the package size, this is also a property of *Postage*. If you don't need it, leave it at 0.

2.7.1 Configuration

You can change the displayed units, e.g. if someone insists in obsolete non-metric units:

```
PLATA_SHIPPING_WEIGHT_UNIT = 'g'
PLATA_SHIPPING_LENGTH_UNIT = 'mm'
```

2.8 Database migrations

Plata does not include database migrations. The reason for that is that Django does not support the concept of lazy foreign keys. Because you're free to choose any model as product for Plata (configurable using `PLATA_SHOP_PRODUCT`) we do not know beforehand, how the database constraints for the order item product foreign key should look beforehand. Because of that it's easier for everyone to just not include any migrations and instead provide instructions in the release notes when a database migration has to be performed.

Note: This isn't strictly true anymore now that Django supports swappable models. This guide has been written in 2012 long before Django officially supported a swappable user model.

The following issues on Github will shed further light upon this issue:

- https://github.com/matthiask/plata/issues/33
- https://github.com/matthiask/plata/issues/27
- https://github.com/matthiask/plata/issues/26

Note: Despite not bundling database migrations, using them is very much recommended and officially supported. Any pitfalls will be documented here and in the release notes.

2.8.1 Configuring migrations for Plata

Since the migrations are project-specific, the migration files should be added to your apps, and not to Plata itself. This can be achieved using the following setting:

```
MIGRATION_MODULES = {
    'contact': 'yourapp.migrate.contact',
    'discount': 'yourapp.migrate.discount',
    'stock': 'yourapp.migrate.stock',  # If you're using stock tracking
    'shop': 'yourapp.migrate.shop',
}
```

You'll have to add a folder `migrate` in your app containing only an empty `__init__.py` file for these instructions to work. After that run `makemigrations` on each Plata app and you're hopefully done.

Note: You can use any other name for the `migrate` folder **except for** `migrations`.

2.9 Examples

Plata's repository contains a few example projects, which are used both to demonstrate specific features and allow experimentation.

The purpose of each example project will be briefly described in the following.

2.9.1 `simple` example

This example strives to be the simplest shop implementation with Plata while still following best practice.

2.9.2 `custom` example

The custom example demonstrates how the bundled contact model might be replaced with a custom model – in this case, the replacement model only has one address record, not two addresses, one for shipping and one for billing.

2.9.3 `staggered` example

This example demonstrates how a staggered (or "tiered") price scheme might be implemented outside of Plata's core.

2.9.4 `oneprice` example

This example demonstrates how a product model might be made even simpler: By extending `ProductBase` and `PriceBase` at the same time, all product related information is kept in one Django model. This (obviously) means that each product can only have one price.

2.9.5 `generic` example

Plata accepts only one product model. But you might want to sell widely differing stuff, like physical things and downloads, that need different models. This example shows how to setup a product model with a generic relationship to the real product.

The data entry is somewhat complicated – first create some "Things" and "Downloads", then edit the "Products" for their prices.

This example also uses the `CountryGroup` from `plata.shipping` to implement prices that depend on the customer's country.

2.9.6 setup for examples

- make sure that `examples/manage.py` is configured to use your example of choice, e.g. 'simple'.
- setup basic Django models: `./manage.py migrate`
- prepare shop models (important: all at once, otherwise you'll run into a dependency hell): `./manage.py makemigrations shop contact discount simple`
- setup shop models: `./manage.py migrate`
- create an user: `./manage.py createsuperuser` (the examples contain no user management and depend on admin staff users.)
- create tax classes, products, discounts etc. at the admin site

If you want to try different examples, you must delete the migrations from shop, contact and discount!

For your convenience, we included two (bash) shell scripts, to be called from within the examples directory of a git checkout.

- *reset-migrations.sh* lets you delete the migrations from all examples and all plata modules.
- *startexample.sh* automates the necessary migrations setup and starts one example.

There are some checks and prompts to avoid damage, but of course we can't be held responsible for anything that happens.

CHAPTER 3

API Documentation

3.1 Core

plata.**product_model**()
Return the product model defined by the PLATA_SHOP_PRODUCT setting.

plata.**shop_instance**()
This method ensures that all views and URLs are properly loaded, and returns the centrally instantiated plata.shop.views.Shop object.

plata.**stock_model**()
Return the stock transaction model definded by the PLATA_STOCK_TRACKING_MODEL setting or None in case stock transactions are turned off.

3.2 Products

Product model base implementation – you do not need to use this

It may save you some typing though.

class plata.product.models.**ProductBase**(**args*, ***kwargs*)
Product models must have two methods to be usable with Plata:

- get_price: Return a price instance
- handle_order_item: Fill in fields on the order item from the product, i.e. the name and the stock keeping unit.

get_price(*currency=None*, *orderitem=None*)
This method is part of the public, required API of products. It returns either a price instance or raises a DoesNotExist exception.

If you need more complex pricing schemes, override this method with your own implementation.

handle_order_item(*orderitem*)
This method has to ensure that the information on the order item is sufficient for posteriority. Old orders should always be complete even if the products have been changed or deleted in the meantime.

3.2.1 Product extensions

Exact, transactional stock tracking for Plata

Follow these steps to enable this module:

- Ensure your product model has an `items_in_stock` field with the following definiton:

```
items_in_stock = models.IntegerField(default=0)
```

- Add `'plata.product.stock'` to `INSTALLED_APPS`.
- Set `PLATA_STOCK_TRACKING = True` to enable stock tracking in the checkout and payment processes.
- Optionally modify your add-to-cart forms on product detail pages to take into account `items_in_stock`.

class plata.product.stock.models.**Period**(**args*, ***kwargs*)
A period in which stock changes are tracked

You might want to create a new period every year and create initial amount transactions for every variation.

`StockTransaction.objects.open_new_period` does this automatically.

class plata.product.stock.models.**StockTransaction**(**args*, ***kwargs*)
Stores stock transactions transactionally :-)

Stock transactions basically consist of a product variation reference, an amount, a type and a timestamp. The following types are available:

- `StockTransaction.INITIAL`: Initial amount, used when filling in the stock database
- `StockTransaction.CORRECTION`: Use this for any errors
- `StockTransaction.PURCHASE`: Product purchase from a supplier
- `StockTransaction.SALE`: Sales, f.e. through the webshop
- `StockTransaction.RETURNS`: Returned products (i.e. from lending)
- `StockTransaction.RESERVATION`: Reservations
- `StockTransaction.INCOMING`: Generic warehousing
- `StockTransaction.OUTGOING`: Generic warehousing
- `StockTransaction.PAYMENT_PROCESS_RESERVATION`: Product reservation during payment process

Most of these types do not have a significance to Plata. The exceptions are:

- `INITIAL` transactions are created by `open_new_period`
- `SALE` transactions are created when orders are confirmed
- `PAYMENT_PROCESS_RESERVATION` transactions are created by payment modules which send the user to a different domain for payment data entry (f.e. PayPal). These transactions are also special in that they are only valid for 15 minutes. After 15 minutes, other customers are able to put the product in their cart and proceed to checkout again. This time period is a security measure against customers buying products at the same time which cannot be delivered afterwards because stock isn't available.

plata.product.stock.models.**validate_order_stock_available**(*order*)
Check whether enough stock is available for all selected products, taking into account payment process reservations.

3.3 Discounts

class plata.discount.models.**AppliedDiscount**(**args*, ***kwargs*)
Stores an applied discount, so that deletion of discounts does not affect orders.

exception DoesNotExist

exception MultipleObjectsReturned

class plata.discount.models.**AppliedDiscountManager**
Default manager for the `AppliedDiscount` model

remaining(*order=None*)
Calculate remaining discount excl. tax

Can either be used as related manager:

```
order.applied_discounts.remaining()
```

or directly:

```
AppliedDiscount.objects.remaining(order)
```

class plata.discount.models.**Discount**(*id*, *name*, *type*, *value*, *currency*, *tax_class*, *config*, *code*, *is_active*, *valid_from*, *valid_until*, *allowed_uses*, *used*)

exception DoesNotExist

exception MultipleObjectsReturned

add_to(*order*, *recalculate=True*)
Add discount to passed order

Removes the previous discount if a discount with this code has already been added to the order before.

validate(*order*)
Validate whether this discount can be applied on the given order

class plata.discount.models.**DiscountBase**(**args*, ***kwargs*)
Base class for discounts and applied discounts

CONFIG_OPTIONS = [(u'all', {u'title': <django.utils.functional.__proxy__ object>}), (
You can add and remove options at will, except for 'all': This option must always be available, and it cannot have any form fields

3.4 Shop

class plata.shop.models.**BillingShippingAddress**(**args*, ***kwargs*)
Abstract base class for all models storing a billing and a shipping address

addresses()
Return a `dict` containing a billing and a shipping address, taking into account the value of the `shipping_same_as_billing` flag

class plata.shop.models.**Order**(**args*, ***kwargs*)

The main order model. Used for carts and orders alike.

CART = 10

Order object is a cart.

CHECKOUT = 20

Checkout process has started.

COMPLETED = 50

Order has been completed. Plata itself never sets this state, it is only meant for use by the shop owners.

CONFIRMED = 30

Order has been confirmed, but it not (completely) paid for yet.

exception DoesNotExist

exception MultipleObjectsReturned

PAID = 40

Order has been completely paid for.

PENDING = 35

For invoice payment methods, when waiting for the money

VALIDATE_ALL = 100

This should not be used while registering a validator, it's mostly useful as an argument to `validate()` when you want to run all validators.

VALIDATE_BASE = 10

This validator is always called; basic consistency checks such as whether the currencies in the order match should be added here.

VALIDATE_CART = 20

A cart which fails the criteria added to the `VALIDATE_CART` group isn't considered a valid cart and the user cannot proceed to the checkout form. Stuff such as stock checking, minimal order total checking, or maximal items checking might be added here.

balance_remaining

Returns the balance which needs to be paid by the customer to fully pay this order. This value is not necessarily the same as the order total, because there can be more than one order payment in principle.

discount

Returns the discount total.

discount_remaining

Remaining discount amount excl. tax

is_confirmed()

Returns `True` if this order has already been confirmed and therefore cannot be modified anymore.

items_in_order()

Returns the item count in the order

This is different from `order.items.count()` because it counts items, not distinct products.

modify_item(*product*, *relative=None*, *absolute=None*, *recalculate=True*, *data=None*, *item=None*, *force_new=False*)

Updates order with the given product

- `relative` or `absolute`: Add/subtract or define order item amount exactly
- `recalculate`: Recalculate order after cart modification (defaults to `True`)

- `data`: Additional data for the order item; replaces the contents of the JSON field if it is not `None`. Pass an empty dictionary if you want to reset the contents.
- `item`: The order item which should be modified. Will be automatically detected using the product if unspecified.
- `force_new`: Force the creation of a new order item, even if the product exists already in the cart (especially useful if the product is configurable).

Returns the `OrderItem` instance; if quantity is zero, the order item instance is deleted, the `pk` attribute set to `None` but the order item is returned anyway.

order_id
Returns `_order_id` (if it has been set) or a generic ID for this order.

recalculate_total(*save=True*)
Recalculates totals, discounts, taxes.

classmethod register_validator(*validator*, *group*)
Registers another order validator in a validation group

A validator is a callable accepting an order (and only an order).

There are several types of order validators:

- Base validators are always called
- Cart validators: Need to validate for a valid cart
- Checkout validators: Need to validate in the checkout process

reload()
Return this order instance, reloaded from the database

Used f.e. inside the payment processors when adding new payment records etc.

save(**args*, ***kwargs*)
Sequential order IDs for completed orders.

shipping
Returns the shipping cost, with or without tax depending on this order's `price_includes_tax` field.

subtotal
Returns the order subtotal.

tax
Returns the tax total for this order, meaning tax on order items and tax on shipping.

update_status(*status*, *notes*)
Update the order status

validate(*group*)
Validates this order

The argument determines which order validators are called:

- `Order.VALIDATE_BASE`
- `Order.VALIDATE_CART`
- `Order.VALIDATE_CHECKOUT`
- `Order.VALIDATE_ALL`

class plata.shop.models.**OrderItem**(**args*, ***kwargs*)
Single order line item

exception DoesNotExist

exception MultipleObjectsReturned

class plata.shop.models.**OrderPayment**(**args*, ***kwargs*)
Order payment

Stores additional data from the payment interface for analysis and accountability.

exception DoesNotExist

exception MultipleObjectsReturned

class plata.shop.models.**OrderStatus**(**args*, ***kwargs*)
Order status

Stored in separate model so that the order status changes stay visible for analysis after the fact.

exception DoesNotExist

exception MultipleObjectsReturned

class plata.shop.models.**PriceBase**(**args*, ***kwargs*)
Price for a given product, currency, tax class and time period

Prices should not be changed or deleted but replaced by more recent prices. (Deleting old prices does not hurt, but the price history cannot be reconstructed anymore if you'd need it.)

The concrete implementation needs to provide a foreign key to the product model.

handle_order_item(*item*)
Set price data on the `OrderItem` passed

class plata.shop.models.**TaxClass**(**args*, ***kwargs*)
Tax class, storing a tax rate

TODO informational / advisory currency or country fields?

exception DoesNotExist

exception MultipleObjectsReturned

plata.shop.models.**validate_order_currencies**(*order*)
Check whether order contains more than one or an invalid currency

3.4.1 Order processors

class plata.shop.processors.**ApplyRemainingDiscountToShippingProcessor**(*shared_state*)
Apply the remaining discount to the shipping (if shipping is non-zero and there are any remaining discounts left)

class plata.shop.processors.**DiscountProcessor**(*shared_state*)
Apply all discounts which do not act as a means of payment but instead act on the subtotal

class plata.shop.processors.**FixedAmountShippingProcessor**(*shared_state*)
Set shipping costs to a fixed value. Uses `PLATA_SHIPPING_FIXEDAMOUNT`. If you have differing needs you should probably implement your own shipping processor (and propose it for inclusion if you like) instead of extending this one.

```
PLATA_SHIPPING_FIXEDAMOUNT = {
    'cost': Decimal('8.00'),
    'tax': Decimal('19.6'),
    }
```

class plata.shop.processors.**InitializeOrderProcessor**(*shared_state*)
Zero out all relevant order values and calculate line item prices excl. tax.

class plata.shop.processors.**ItemSummationProcessor**(*shared_state*)
Sum up line item prices, discounts and taxes.

class plata.shop.processors.**MeansOfPaymentDiscountProcessor**(*shared_state*)
Apply all discounts which act as a means of payment.

class plata.shop.processors.**OrderSummationProcessor**(*shared_state*)
Sum up order total by adding up items and shipping totals.

process(*order*, *items*)
The value must be quantized here, because otherwise f.e. the payment modules will be susceptible to rounding errors giving f.e. missing payments of 0.01 units.

class plata.shop.processors.**ProcessorBase**(*shared_state*)
Order processor class base. Offers helper methods for order total aggregation and tax calculation.

add_tax_details(*tax_details*, *tax_rate*, *price*, *discount*, *tax_amount*)
Add tax details grouped by tax_rate. Especially useful if orders potentially use more than one tax class. These values are not used for the order total calculation – they are only needed to show the tax amount for different tax rates if this is necessary for your invoices.

- `tax_details`: The tax details dict, most often stored as `order.data['tax_details'] = tax_details.items()`
- `tax_rate`: The tax rate of the current entry
- `price`: The price excl. tax
- `discount`: The discount amount (will be subtracted from the price before applying the tax)
- `tax_amount`: The exact amount; a bit redundant because this could be calculated using the values above as well

See the taxes documentation or the standard invoice PDF generation code if you need to know more about the use of these values.

process(*order*, *items*)
This is the method which must be implemented in order processor classes.

split_cost(*cost_incl_tax*, *tax_rate*)
Split a cost incl. tax into the part excl. tax and the tax

class plata.shop.processors.**TaxProcessor**(*shared_state*)
Calculate taxes for every line item and aggregate tax details.

class plata.shop.processors.**ZeroShippingProcessor**(*shared_state*)
Set shipping costs to zero.

3.4.2 Template tags

plata.shop.templatetags.plata_tags.**form_errors**(*parser*, *token*)
Show all form and formset errors:

```
{% form_errors form formset1 formset2 %}
```

Silently ignores non-existant variables.

plata.shop.templatetags.plata_tags.**form_item**(*item*, *additional_classes=None*)
Helper for easy displaying of form items:

```
{% for field in form %}{% form_item field %}{% endfor %}
```

plata.shop.templatetags.plata_tags.**form_item_plain**(*item*, *additional_classes=None*)
Helper for easy displaying of form items without any additional tags (table cells or paragraphs) or labels:

```
{% form_item_plain field %}
```

plata.shop.templatetags.plata_tags.**form_items**(*form*)
Render all form items:

```
{% form_items form %}
```

plata.shop.templatetags.plata_tags.**load_plata_context**(*context*)
Conditionally run plata's context processor using {% load_plata_context %}

Rather than having the overheads involved in globally adding it to TEMPLATE_CONTEXT_PROCESSORS.

plata.shop.templatetags.plata_tags.**quantity_ordered**(*product*, *order*)
e.g. {% if product|quantity_ordered:plata.order > 0 %} ... {% endif %}

3.4.3 Signals

plata.shop.signals.**contact_created = <django.dispatch.dispatcher.Signal object>**
Emitted upon contact creation. Receives the user and contact instance and the new password in cleartext.

plata.shop.signals.**order_confirmed = <django.dispatch.dispatcher.Signal object>**
Emitted upon order confirmation. Receives an order instance.

plata.shop.signals.**order_paid = <django.dispatch.dispatcher.Signal object>**
Emitted when an order has been completely paid for. Receives the order and payment instances and the remaining discount amount excl. tax, if there is any.

3.4.4 Notifications

Even though these shop signal handlers might be useful, you might be better off writing your own handlers for the three important signals:

- `contact_created`: A new contact has been created during the checkout process
- `order_confirmed`: The order has been confirmed, a payment method has been selected
- `order_paid`: The order is fully paid

A real-world example follows:

```
from django.utils.translation import activate

from plata.shop import notifications, signals as shop_signals


class EmailHandler(notifications.BaseHandler):
    ALWAYS = ['shopadmin@example.com']
    SHIPPING = ['warehouse@example.com']

    def __call__(self, sender, order, **kwargs):
        cash_on_delivery = False
        try:
```

(continues on next page)

(continued from previous page)

```
            if (order.payments.all()[0].payment_module_key == 'cod'):
                cash_on_delivery = True
        except:
            pass

        if order.language_code:
            activate(order.language_code)

        invoice_message = self.create_email_message(
            'plata/notifications/order_paid.txt',
            order=order,
            **kwargs)
        invoice_message.attach(order.order_id + '.pdf',
            self.invoice_pdf(order), 'application/pdf')
        invoice_message.to.append(order.email)
        invoice_message.bcc.extend(self.ALWAYS)

        packing_slip_message = self.create_email_message(
            'plata/notifications/packing_slip.txt',
            order=order,
            **kwargs)
        packing_slip_message.attach(
            order.order_id + '-LS.pdf',
            self.packing_slip_pdf(order),
            'application/pdf')
        packing_slip_message.to.extend(self.ALWAYS)

        if cash_on_delivery:
            invoice_message.bcc.extend(self.SHIPPING)
        else:
            packing_slip_message.to.extend(self.SHIPPING)

        invoice_message.send()
        packing_slip_message.send()

shop_signals.contact_created.connect(
    notifications.ContactCreatedHandler(),
    weak=False)
shop_signals.order_paid.connect(
    EmailHandler(),
    weak=False)
```

class plata.shop.notifications.**ContactCreatedHandler**(*always_to=None*, *always_bcc=None*)

Send an e-mail message to a newly created contact, optionally BCC'ing the addresses passed as `always_bcc` upon handler initialization.

Usage:

```
signals.contact_created.connect(
    ContactCreatedHandler(),
    weak=False)
```

or:

```
signals.contact_created.connect(
    ContactCreatedHandler(always_bcc=['owner@example.com']),
```

(continues on next page)

(continued from previous page)

```
    weak=False)
```

class plata.shop.notifications.**SendInvoiceHandler**(*always_to=None*, *always_bcc=None*)

Send an e-mail with attached invoice to the customer after successful order completion, optionally BCC'ing the addresses passed as `always_bcc` to the handler upon initialization.

Usage:

```
signals.order_paid.connect(
    SendInvoiceHandler(always_bcc=['owner@example.com']),
    weak=False)
```

class plata.shop.notifications.**SendPackingSlipHandler**(*always_to=None*, *always_bcc=None*)

Send an e-mail with attached packing slip to the addresses specified upon handler initialization. You should pass at least one address in either the `always_to` or the `always_bcc` argument, or else the e-mail will go nowhere.

Usage:

```
signals.order_paid.connect(
    SendPackingSlipHandler(always_to=['warehouse@example.com']),
    weak=False)
```

3.5 Payment

class plata.payment.modules.base.**ProcessorBase**(*shop*)

Payment processor base class

already_paid(*order*, *request=None*)

Handles the case where a payment module is selected but the order is already completely paid for (f.e. because an amount discount has been used which covers the order).

Does nothing if the order **status** is `PAID` already.

clear_pending_payments(*order*)

Clear pending payments

create_pending_payment(*order*)

Create a pending payment

create_transactions(*order*, *stage*, ***kwargs*)

Create transactions for all order items. The real work is offloaded to `StockTransaction.objects.bulk_create`.

default_name = u'unnamed'

Human-readable name for this payment module. You may use i18n here.

enabled_for_request(*request*)

Decides whether this payment module is available for a given request.

Defaults to `True`. If you need to disable payment modules for certain visitors or group of visitors, that is the method you are searching for.

get_urls()

Defines URLs for this payment processor

Note that these URLs are added directly to the shop views URLconf without prefixes. It is your responsability to namespace these URLs so they don't clash with shop views and other payment processors.

key = u'unnamed'
Safe key for this payment module (shouldn't contain special chars, spaces etc.)

name
Returns name of this payment module suitable for human consumption

Defaults to `default_name` but can be overridden by placing an entry in `PLATA_PAYMENT_MODULE_NAMES`. Example:

```
PLATA_PAYMENT_MODULE_NAMES = {
    'paypal': _('Paypal and credit cards'),
    }
```

order_paid(*order*, *payment=None*, *request=None*)
Call this when the order has been fully paid for.

This method does the following:

- Sets order status to `PAID`.
- Calculates the remaining discount amount (if any) and calls the `order_paid` signal.
- Clears pending payments which aren't interesting anymore anyway.

process_order_confirmed(*request*, *order*)
This is the initial entry point of payment modules and is called when the user has selected a payment module and accepted the terms and conditions of the shop.

Must return a response which is presented to the user, i.e. a form with hidden values forwarding the user to the PSP or a redirect to the success page if no further processing is needed.

urls
Returns URLconf definitions used by this payment processor

This is especially useful for processors offering server-to-server communication such as Paypal's IPN (Instant Payment Notification) where Paypal communicates payment success immediately and directly, without involving the client.

Define your own URLs in `get_urls`.

3.5.1 Cash on delivery

Payment module for cash on delivery handling

Automatically completes every order passed.

3.5.2 Check support

Payment module for check/transfer

Author: jpbraun@mandriva.com

Configuration: PLATA_PAYMENT_CHECK_NOTIFICATIONS

3.5.3 PayPal support

Payment module for PayPal integration

Needs the following settings to work correctly:

```
PAYPAL = {
    'BUSINESS': 'yourbusiness@paypal.com',
    'LIVE': True, # Or False
    }
```

3.5.4 Postfinance support

Payment module for Postfinance integration

Needs the following settings to work correctly:

```
POSTFINANCE = {
    'PSPID': 'your_shop_id',
    'LIVE': True, # Or False
    'SHA1_IN': 'yourhash',
    'SHA1_OUT': 'yourotherhash',
    }
```

3.5.5 Ogone support

Payment module for Ogone integration

Needs the following settings to work correctly:

```
OGONE = {
    'PSPID': 'your_shop_id',
    'LIVE': True, # Or False
    'SHA1_IN': 'yourhash',
    'SHA1_OUT': 'yourotherhash',
    }
```

3.5.6 Datatrans support

Payment module for Datatrans integration

Needs the following settings to work correctly:

```
DATATRANS = {
    'MERCHANT_ID': '1000000000000',
    'LIVE': False
    }
```

3.5.7 Billogram support

3.5.8 Payson support

3.5.9 PagSeguro support

Pagseguro payment module for django-plata Authors: alexandre@mandriva.com.br, jpbraun@mandriva.com Date: 03/14/2012

3.5.10 Stripe support

3.6 Shipping

3.7 Views

class plata.shop.views.**Shop**(*contact_model*, *order_model*, *discount_model*, *default_currency=None*, ***kwargs*)

Plata's view and shop processing logic is contained inside this class.

Shop needs a few model classes with relations between them:

- Contact model linking to Django's auth.user
- Order model with order items and an applied discount model
- Discount model
- Default currency for the shop (if you do not override default_currency in your own Shop subclass)

Example:

```
shop_instance = Shop(Contact, Order, Discount)

urlpatterns = [
    url(r'^shop/', include(shop_instance.urls)),
]
```

base_template = u'base.html'

The base template used in all default checkout templates

cart(*request*, *order*)

Shopping cart view

checkout(*request*, *order*)

Handles the first step of the checkout process

checkout_form(*request*, *order*)

Returns the address form used in the first checkout step

confirmation(*request*, *order*)

Handles the order confirmation and payment module selection checkout step

Hands off processing to the selected payment module if confirmation was successful.

confirmation_form(*request*, *order*)

Returns the confirmation and payment module selection form

contact_from_user(*user*)
Return the contact object bound to the current user if the user is authenticated. Returns `None` if no contact exists.

create_order_for_user(*user*, *request=None*)
Creates and returns a new order for the given user.

default_currency(*request=None*)
Return the default currency for instantiating new orders

Override this with your own implementation if you have a multi-currency shop with auto-detection of currencies.

discounts(*request*, *order*)
Handles the discount code entry page

discounts_form(*request*, *order*)
Returns the discount form

get_context(*request*, *context*, ***kwargs*)
Helper method returning a context dict. Override this if you need additional context variables.

get_payment_modules(*request=None*)
Import and return all payment modules defined in `PLATA_PAYMENT_MODULES`

If request is given only applicable modules are loaded.

order_from_request(*request*, *create=False*)
Instantiate the order instance for the current session. Optionally creates a new order instance if `create=True`.

Returns `None` if unable to find an offer.

order_new(*request*)
Forcibly create a new order and redirect user either to the frontpage or to the URL passed as `next` GET parameter

order_payment_failure(*request*)
Handles order payment failures

order_payment_pending(*request*)
Handles order successes for invoice payments where payment is still pending.

order_success(*request*)
Handles order successes (e.g. when an order has been successfully paid for)

price_includes_tax(*request=None*)
Return if the shop should show prices including tax

This returns the PLATA_PRICE_INCLUDES_TAX settings by default and is meant to be overridden by subclassing the Shop.

redirect(*url_name*, **args*, ***kwargs*)
Hook for customizing the redirect function when used as application content

render(*request*, *template*, *context*)
Helper which just passes everything on to `django.shortcuts.render`

render_cart(*request*, *context*)
Renders the shopping cart

render_cart_empty(*request*, *context*)
Renders a cart-is-empty page

render_checkout(*request*, *context*)
Renders the checkout page

render_confirmation(*request*, *context*)
Renders the confirmation page

render_discounts(*request*, *context*)
Renders the discount code entry page

reverse_url(*url_name*, **args*, ***kwargs*)
Hook for customizing the reverse function

set_order_on_request(*request*, *order*)
Helper method encapsulating the process of setting the current order in the session. Pass `None` if you want to remove any defined order from the session.

urls
Property offering access to the Shop-managed URL patterns

user_is_authenticated(*user*)
Overwrite this for custom authentication check. This is needed to support lazysignup

plata.shop.views.**cart_not_empty**(*order*, *shop*, *request*, ***kwargs*)
Redirect to cart if later in checkout process and cart empty

plata.shop.views.**checkout_process_decorator**(**checks*)
Calls all passed checkout process decorators in turn:

```
@checkout_process_decorator(order_already_confirmed,
    order_cart_validates)
```

All checkout process decorators are called with the order, the shop instance and the request as keyword arguments. In the future, additional keywords might be added, your decorators should accept `**kwargs` as well for future compatibility.

plata.shop.views.**order_already_confirmed**(*order*, *shop*, *request*, ***kwargs*)
Redirect to confirmation or already paid view if the order is already confirmed

plata.shop.views.**order_cart_validates**(*order*, *shop*, *request*, ***kwargs*)
Redirect to cart if stock is insufficient and display an error message

plata.shop.views.**order_cart_warnings**(*order*, *shop*, *request*, ***kwargs*)
Show warnings in cart, but don't redirect (meant as a replacement for `order_cart_validates`, but usable on the cart view itself)

plata.shop.views.**user_is_authenticated**(*order*, *shop*, *request*, ***kwargs*)
ensure the user is authenticated and redirect to checkout if not

3.8 Context processors

plata.context_processors.**plata_context**(*request*)
Adds a few variables from Plata to the context if they are available:

- `plata.shop`: The current `plata.shop.views.Shop` instance
- `plata.order`: The current order
- `plata.contact`: The current contact instance
- `plata.price_includes_tax`: Whether prices include tax or not

3.9 Fields

plata.fields.**CurrencyField**(**moreargs*, ***morekwargs*)
Field offering all defined currencies

class plata.fields.**JSONField**(*verbose_name=None*, *name=None*, *primary_key=False*, *max_length=None*, *unique=False*, *blank=False*, *null=False*, *db_index=False*, *rel=None*, *default=<class django.db.models.fields.NOT_PROVIDED>*, *editable=True*, *serialize=True*, *unique_for_date=None*, *unique_for_month=None*, *unique_for_year=None*, *choices=None*, *help_text=u''*, *db_column=None*, *db_tablespace=None*, *auto_created=False*, *validators=[]*, *error_messages=None*)
TextField which transparently serializes/unserializes JSON objects

See: http://www.djangosnippets.org/snippets/1478/

from_db_value(*value*, *expression*, *connection*, *context*)
Convert the input JSON value into python structures, raises django.core.exceptions.ValidationError if the data can't be converted.

get_prep_value(*value*)
Convert our JSON object to a string before we save

to_python(*value*)
Convert our string value to JSON after we load it from the DB

value_to_string(*obj*)
Extract our value from the passed object and return it in string form

3.10 Utilities

plata.utils.**jsonize**(*v*)
Convert the discount configuration into a state in which it can be stored inside the JSON field.

Some information is lost here; f.e. we only store the primary key of model objects, so you have to remember yourself which objects are meant by the primary key values.

3.11 Reporting

3.11.1 Order reports

plata.reporting.order.**invoice_pdf**(*pdf*, *order*)
PDF suitable for use as invoice

plata.reporting.order.**packing_slip_pdf**(*pdf*, *order*)
PDF suitable for use as packing slip

3.11.2 Product reports

plata.reporting.product.**product_xls**()
Create a list of all product variations, including stock and aggregated stock transactions (by type)

3.11.3 Reporting views

plata.reporting.views.**invoice**(*request*, **args*, ***kwargs*)
Returns the invoice PDF

plata.reporting.views.**invoice_pdf**(*request*, **args*, ***kwargs*)
Returns the invoice PDF

plata.reporting.views.**packing_slip_pdf**(*request*, **args*, ***kwargs*)
Returns the packing slip PDF

plata.reporting.views.**product_xls**(*request*, **args*, ***kwargs*)
Returns an XLS containing product information

3.12 Settings

The settings here should be accessed using `plata.settings.<KEY>`. All settings can be overridden by putting them in your standard `settings.py` module.

plata.default_settings.**CURRENCIES = (u'CHF', u'EUR', u'USD', u'CAD')**
All available currencies. Use ISO 4217 currency codes in this list only.

plata.default_settings.**CURRENCIES_WITHOUT_CENTS = (u'JPY', u'KRW')**
If you use currencies that don't have a minor unit (zero-decimal currencies) At the moment only relevant to Stripe payments.

plata.default_settings.**PLATA_ORDER_PROCESSORS = [u'plata.shop.processors.InitializeOrderPro**
List of order processors

Plata does not check whether the selection makes any sense. This is your responsibility.

plata.default_settings.**PLATA_PAYMENT_MODULES = [u'plata.payment.modules.cod.PaymentProcesso**
Activated payment modules

plata.default_settings.**PLATA_PAYMENT_MODULE_NAMES = {}**
Override payment module names without modifying the payment module code

The key in this dictionary should use the `key` variable of the respective payment module.

Example:

```
PLATA_PAYMENT_MODULE_NAMES = {
    'paypal': 'PayPal and Credit Cards',
}
```

plata.default_settings.**PLATA_PDF_FONT_NAME = u''**
Custom font for PDF generation

plata.default_settings.**PLATA_PRICE_INCLUDES_TAX = True**
Are prices shown with tax included or not? (Defaults to `True`) Please note that this setting is purely presentational and has no influence on the values stored in the database.

plata.default_settings.**PLATA_REPORTING_ADDRESSLINE = u''**
PDF address line

plata.default_settings.**PLATA_REPORTING_STATIONERY = u'pdfdocument.elements.ExampleStationer**
Stationery for invoice and packing slip PDF generation

plata.default_settings.**PLATA_SHIPPING_FIXEDAMOUNT = {u'cost': Decimal('8.00'), u'tax': De**
`FixedAmountShippingProcessor` example configuration

The cost must be specified with tax included.

plata.default_settings.**PLATA_SHIPPING_WEIGHT_UNIT = u'g'**
`shipping.Postage` configuration change this if you insist in obsolete non-metric units

plata.default_settings.**PLATA_SHOP_PRODUCT = u'product.Product'**
Target of order item product foreign key (Defaults to `'product.Product'`)

plata.default_settings.**PLATA_STOCK_TRACKING = False**
Transactional stock tracking

`'plata.product.stock'` has to be included in `INSTALLED_APPS` for this to work.

plata.default_settings.**PLATA_ZIP_CODE_LABEL = <django.utils.functional.__proxy__ object>**
Since ZIP code is far from universal, and more an L10N than I18N issue:

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

P