django SHOP

Release 0.2.1.dev0

May 26, 2016

1 User Manual 1

2 How to 11

3 Advanced how to 19

4 Reference 23

5 The name 27

Python Module Index 29

CHAPTER 1

User Manual

The “instructions” :)

1.1 Tutorial

This tutorial is aimed at people new to django SHOP but already familiar with django. If you aren’t yet, reading their excellent django tutorial is highly recommended.

The steps outlined in this tutorial are meant to be followed in order.

1.1.1 Installation

- Install from pypi

```
pip install django-shop
```

- Add `'shop'` to your *INSTALLED_APPS*
- Add the shop to your *urls.py*

```
(r'^shop/', include('shop.urls')),
```

1.1.2 Defining your products

The first thing a shop should do is define its products to sell. While some other shop solutions do not require you to, in django SHOP you must write a django model (or a set of django models) to represent your products.

Roughly, this means you’ll need to create a django model subclassing *`shop.models.Product`*, add a `Meta` class to it, and register it with the admin, just like you would do with any other django model.

More information can be found in the How to create a product section.

1.1.3 Shop rules: cart modifiers

Cart modifiers are simple python objects that encapsulate all the pricing logic from your specific shop, such as rebates, taxes, coupons, deals of the week...

Creating a new cart modifier is an easy task: simply create a python object subclassing `shop.cart.cart_modifiers_base.BaseCartModifier`, and override either its `get_extra_cart_item_price_field()` or its `get_extra_cart_price_field()`, depending on whether your "rule" applies to the whole cart (like taxes for example) or to a single item in your cart (like "buy two, get one free" rebates).

Theses methods receive either the cart object or the cart item, and need only return a tuple of the form `(description, price_difference)`.

More in-depth information can be found in the How to create a Cart modifier section.

1.1.4 Shipping backends

1.1.5 Payment backends

1.1.6 More plugins?

You can find more plugins or share your own plugin with the world on the django SHOP website

Lots of functionality in django SHOP was left to implement as plugins and extensions, checking this resource for extra functionality is highly recommended before starting a new project!

1.2 Getting started

1.2.1 Installation

Here's the 1 minute guide to getting started with django SHOP.

1. Create a normal Django project (we'll call it `myshop` for now):

```
django-admin startproject example
cd example; django-admin startapp myshop
```

2. You'll want to use virtualenv:

```
virtualenv . ; source bin/activate
pip install south
pip install django-shop
pip install jsonfield
```

3. Go to your settings.py and configure your DB like the following, or anything matching your setup:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'test.sqlite',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    }
}
```

4. Add the following stuff to middlewares:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
] # <-- Notice how it's a square bracket (a list)? It makes life easier.
```

5. Obviously, you need to add shop and myshop to your INSTALLED_APPS too:

```
INSTALLED_APPS = [
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    'django.contrib.admindocs',
    'polymorphic', # We need polymorphic installed for the shop
    'south',
    'shop', # The django SHOP application
    'shop.addressmodel', # The default Address and country models
    'myshop', # the project we just created
]
```

6. Make the urls.py contain the following:

```
from shop import urls as shop_urls # <-- Add this at the top

# Other stuff here

urlpatterns = patterns('',
    # Example:
    #(r'^example/', include('example.foo.urls')),
    # Uncomment the admin/doc line below to enable admin documentation:
    (r'^admin/doc/', include('django.contrib.admindocs.urls')),
    # Uncomment the next line to enable the admin:
    (r'^admin/', include(admin.site.urls)),
    (r'^shop/', include(shop_urls)), # <-- That's the important bit
    # You can obviously mount this somewhere else
)
```

7. Most of the stuff you'll have to do is styling and template work, so go ahead and create a templates directory in your project:

```
cd example/myshop; mkdir -p templates/myshop
```

8. Lock and load:

```
cd .. ; python manage.py syncdb --all ; python manage.py migrate --fake
python manage.py runserver
```

9. Point your browser and marvel at the absence of styling:

```
x-www-browser localhost:8000/shop
```

You now have a running but very empty django SHOP installation.

1.2.2 Adding a custom product

Having a shop running is a good start, but you'll probably want to add at least one product class that you can use to sell to clients!

The process is really simple: you simply need to create a class representing your object in your project's `models.py`. Let's start with a very simple model describing a book:

```
from shop.models import Product
from django.db import models

class Book(Product):
    # The author should probably be a foreign key in the real world, but
    # this is just an example
    author = models.CharField(max_length=255)
    cover_picture = models.ImageField(upload_to='img/book')
    isbn = models.CharField(max_length=255)

    class Meta:
        ordering = ['author']
```

Note: The only limitation is that your product subclass must define a `Meta` class.

Like a normal Django model, you might want to register it in the admin interface to allow for easy editing by your admin users. In an `admin.py` file:

```
from django.contrib import admin

from models import Book

admin.site.register(Book)
```

That's it!

1.2.3 Adding taxes

Adding tax calculations according to local regulations is also something that you will likely have to do. It is relatively easy as well: create a new file in your project, for example `modifiers.py`, and add the following:

```
import decimal

from shop.cart.cart_modifiers_base import BaseCartModifier

class Fixed7PercentTaxRate(BaseCartModifier):
    """
    This will add 7% of the subtotal of the order to the total.

    It is of course not very useful in the real world, but this is an
    example.
    """

    def get_extra_cart_price_field(self, cart, request):
        taxes = decimal.Decimal('0.07') * cart.subtotal_price
        to_append = ('Taxes total', taxes)
        return to_append
```

You can now use this newly created tax modifier in your shop! To do so, simply add the class to the list of cart modifiers defined in your `settings.py` file:

```
SHOP_CART_MODIFIERS = ['myshop.modifiers.Fixed7PercentTaxRate']
```

Restart your server, and you should now see that a cart's total is dynamically augmented to reflect this new rule.

You can implement many other types of rules by overriding either this method or other methods defined in `BaseCartModifier`.

Important: Remember that cart modifiers are ordered! Like middlewares, the order in which they are declared in `settings.SHOP_CART_MODIFIERS` matters.

1.3 Templatetags

Django SHOP ships various templatetags to make quick creation of HTML templates easier. In order to use these templatetags you need to load them in your template

```
{% load shop_tags %}
```

1.3.1 Cart

Renders information about the Cart object. This could be used (for example) to show the total amount of items currently in the cart on every page of your shop.

Usage

```
{% load shop_tags %}
{% cart %}
```

In order to define your own template, override the template `shop/templatetags/_cart.html`. The tag adds a variable called `cart` to the context.

1.3.2 Order

Renders information about an Order object.

Usage

```
{% load shop_tags %}
{% order my_order %}
```

In order to define your own template, override the template `shop/templatetags/_order.html`. The tag adds a variable called `order` to the context.

1.3.3 Product

Renders information about all active products in the shop. This is useful if you need to display your products on pages other than just `product_list.html`.

Usage

If no argument is given, the tag will just render all active products. The tag allows an optional argument `objects`. It should be a queryset of *Product* objects. If supplied, the tag will render the given products instead of all active products.

```
{% load shop_tags %}
{% products %}
{% products object_list %}
```

In order to define your own template, override the template `shop/templatetags/_products.html`. The tag adds a variable called `products` to the context.

1.3.4 Filters

priceformat

Renders the float using the `SHOP_PRICE_FORMAT` format. This should be used whenever displaying prices in the templates.

```
{{ product.get_price|priceformat }}
```

1.4 Signals

1.4.1 Order

The *shop.order_signals* module defines signals that are emitted during the checkout process

Warning: Currently, not all signals are emitted inside of django SHOP. This may change in the future.

processing

shop.order_signals.**processing**

Emitted when the `Cart` instance was converted to an `Order`.

Arguments sent with this signal:

sender The `Order` model class

order The `Order` instance

cart The `Cart` instance

payment_selection

shop.order_signals.**payment_selection**

Emitted when the user is shown the “select a payment method” page.

Arguments sent with this signal:

sender The *shop.shipping.api.ShippingAPI* instance

order The `Order` instance

confirmed

`shop.order_signals.`**confirmed**

Emitted when the user has finished placing his order (regardless of the payment success or failure).

Arguments sent with this signal:

sender not defined

order The `Order` instance

Note: This signal is currently not emitted.

completed

`shop.order_signals.`**completed**

Emitted when payment is received for the `Order`. This signal is emitted by the `shop.views.checkout.ThankYouView`.

Arguments sent with this signal:

sender The `ThankYouView` instance

order The `Order` instance

cancelled

`shop.order_signals.`**cancelled**

Emitted if the payment was refused or another fatal problem occurred.

Arguments sent with this signal:

sender not defined

order The `Order` instance

Note: This signal is currently not emitted.

shipped

`shop.order_signals.`**shipped**

Emitted (manually) when the shop clerk or robot shipped the order.

Arguments sent with this signal:

sender not defined

order The `Order` instance

Note: This signal is currently not emitted.

1.5 Contributing

1.5.1 Running tests

It's important to run tests before committing :)

Setting up the environment

We highly suggest you run the tests suite in a clean environment, using a tool such as virtualenv

The following packages are needed for the test suite to run:

- django
- django_polymorphic
- django-classy-tags

Running the following command inside your virtualenv should get you started:

```
pip install django django_polymorphic django-classy-tags
```

Running the tests

Thankfully, we provided a small yet handy script to do it for you! Simply invoke `runtests.sh` on a unix platform and you should be all set.

The test suite should output normally (only ".“‘s), and we try to keep the suite fast (subsecond), so that people can test very often.

Options

While a simple tool, `runtests.sh` provides the following options:

- `--with-coverage` : run the tests using coverage and let the coverage results be displayed in your default browser (run `pip install coverage` beforehand)
- `--with-docs` : run the tests and generate the documentation (the one you're reading right now).

1.5.2 Community

Most of the discussion around django SHOP takes place on IRC (Internet Relay Chat), on the freenode servers in the #django-shop channel.

We also have a mailing list and a google group:

```
http://groups.google.com/group/django-shop
```

1.5.3 Code guidelines

- Like most projects, we try to follow **PEP 8** as closely as possible
- Most pull requests will be rejected without proper unit testing
- Generally we like to discuss new features before they are merged in, but this is a somewhat flexible rule :)

1.5.4 Sending a pull request

We use github for development, and so all code that you would like to see included should follow the following simple workflow:

- Clone django-shop
- Checkout your fork
- Make a feature branch (to make pull requests easier)
- Hack hack, Test test, Commit commit, Test test
- Push your feature branch to your remote (your fork)
- Use the github interface to create a pull request from your branch
- Wait for the community to review your changes. You can hang out with us and ask for feedback on #django-shop (on freenode) in the mean time!
- If some changes are required, please commit to your local feature branch and push the changes to your remote feature branch. The pull request will be updated automagically with your new changes!
- DO NOT add unrelated commits to your branch, since they make the review process more complicated and painful for everybody.

More information can be found on Github itself: http://help.github.com/send-pull-requests/

CHAPTER 2

How to

Various short articles on how to do specific things

2.1 How to create a product

Creating a product in django SHOP is really easy, but requires you to write python code.

2.1.1 Create a model

The first step for you is to create a model to use as a product. We will create an example Book model together:

```
from shop.models import Product
from django.db import models

class Book(Product):
    isbn = models.CharField(max_length=255)
    class Meta: pass
```

Note: Your product subclass must define a `Meta` class. Usually, you will want to do so anyway, to define ordering and verbose names for example.

The following fields are already defined in the *`Product`* superclass:

class `shop.models.`**`Product`**

`name`
The name/title of the product

`slug`
The slug used to refer to this product in the URLs

`active`
Products flagged as not active will not show in the lists

`date_added`
The date at which the product was first created

`last_modified`
A timestamp of when the product was last modified

`unit_price`
The base price for one item

2.1.2 Create a template

Like other objects in Django, you will need to create a template to display your model's contents to the world.

By default, your *`Product`* subclass will use the `shop/product_detail.html` template as a fallback, but will use your own template if you follow Django's naming conventions: `appname/book_detail.html`.

That's all there is to it :)

2.1.3 Using your newly created Product

Your product should behave like a normal Django model in all circumstances. You should register it in admin (and create an admin class for it) like a normal model.

Code wise, the following options are possible to retrieve your newly created model:

```
# This gets your model's instance the normal way, you get both your model's
# fields and the Product fields
o = MyProduct.objects.get(pk=...)

# This is also possible - You retrieve a MyProduct instance, using the
# Product manager
o = Product.objects.get(pk=...)
```

Note: This is possible thanks to the terrific django_polymorphic dependency

2.1.4 Product variations

By design, django SHOP does not include an out of the box solution to handling product variations (colors, sizes...) in order to let implementors create their own unrestricted solutions.

If you want such a pre-made solution for simple cases, we suggest you take a look at the shop_simplevariations "add-on" application.

2.2 How to create a Cart modifier

Cart modifiers are simple plugins that allow you to define rules according to which carts should be modified (and in what order).

Generally, this is how you implement a "bulk rebate" module, for instance.

2.2.1 Writing a simple cart modifier for the whole cart

Let's assume you want to provide a discount of 10% off the cart's total to clients that buy more than 500$ of goods.

This will affect the price of the whole cart. We will therefore override the `get_extra_cart_price_field()` method of `shop.cart.cart_modifiers_base.BaseCartModifier`:

```
from shop.cart.cart_modifiers_base import BaseCartModifier
from decimal import Decimal # We need this because python's float are confusing

class MyModifier(BaseCartModifier):
    """
    An example class that will grant a 10% discount to shoppers who buy
    more than 500 worth of goods.
    """
    def get_extra_cart_price_field(self, cart, request):
        ten_percent = Decimal('10') / Decimal('100')
        # Now we need the current cart total. It's not just the subtotal field
        # because there may be other modifiers before this one
        total = cart.current_total

        if total > Decimal('500'):
            rebate_amount = total * ten_percent
            rebate_amount = - rebate_amount # a rebate is a negative difference
            extra_dict = { 'Rebate': '%s %%' % ten_percent }
            return ('My awesome rebate', rebate_amount)
        else:
            return None # None is no-op: it means "do nothing"

    def get_extra_cart_item_price_field(self, cart, request):
        # Do modifications per cart item here
        label = 'a label'  # to distinguish, which modifier changed the price
        extra_price = Decimal(0)  # calculate addition cost here, can be a negative value
        extra_dict = {}  # an optional Python dictionary serializable as JSON
                         # which can be used to store arbitrary data
        return (label, extra_price, extra_dict)
```

Adding this cart modifier to your SHOP_CART_MODIFIERS setting will enable it, and you should be able to already test that your cart displays a rebate when the total for the order is over 500.

Note: When using `cart.extra_price_fields.append('your label', price)` you might want to use `from django.utils.translation import ugettext as _` for your label in multilingual projects. Please make sure that you use `gettext`

The request object is passed into the methods `get_extra_cart_price_field` and `get_extra_cart_item_price_field`. This object contains the additional temporary attribute `cart_modifier_state`. This is an empty Python dictionary, which can be used to pass arbitrary data from one cart modifier to another one.

2.3 How to create a Payment backend

Payment backends must be listed in settings.SHOP_PAYMENT_BACKENDS

2.3.1 Shop interface

While we could solve this by defining a superclass for all payment backends, the better approach to plugins is to implement inversion-of-control, and let the backends hold a reference to the shop instead.

The reference interface for payment backends is located at

class `shop.payment.api.`**`PaymentAPI`**

Currently, the shop interface defines the following methods:

Common with shipping

`PaymentAPI.`**`get_order`**`(`*request*`)`
Returns the order currently being processed.

Parameters **`request`** – a Django request object

Return type an `Order` instance

`PaymentAPI.`**`add_extra_info`**`(`*order*, *text*`)`
Adds an extra info field to the order (whatever)

Parameters

- **`order`** – an `Order` instance
- **`text`** – a string containing the extra order information

`PaymentAPI.`**`is_order_payed`**`(`*order*`)`
Whether the passed order is fully paid or not

Parameters **`order`** – an `Order` instance

Return type `bool`

`PaymentAPI.`**`is_order_complete`**`(`*order*`)`
Whether the passed order is in a "finished" state

Parameters **`order`** – an `Order` instance

Return type `bool`

`PaymentAPI.`**`get_order_total`**`(`*order*`)`
Returns the order's grand total.

Parameters **`order`** – an `Order` instance

Return type `Decimal`

`PaymentAPI.`**`get_order_subtotal`**`(`*order*`)`
Returns the order's sum of item prices (without taxes or S&H)

Parameters **`order`** – an `Order` instance

Return type `Decimal`

`PaymentAPI.`**`get_order_short_name`**`(`*order*`)`
A short human-readable description of the order

Parameters **`order`** – an `Order` instance

Return type a string with the short name of the order

`PaymentAPI.`**`get_order_unique_id`**`(`*order*`)`
The order's unique identifier for this shop system

Parameters **`order`** – an `Order` instance

Return type the primary key of the `Order` (in the default implementation)

`PaymentAPI.`**`get_order_for_id`**`(`*id*`)`
Returns an `Order` object given a unique identifier (this is the reverse of *`get_order_unique_id()`*)

Parameters **`id`** – identifier for the order

Return type the `Order` object identified by `id`

Specific to payment

`PaymentAPI.`**`confirm_payment`**`(`*order*, *amount*, *transaction_id*, *save=True*`)`

This should be called when the confirmation from the payment processor was called and that the payment was confirmed for a given amount. The processor's transaction identifier should be passed too, along with an instruction to save the object or not. For instance, if you expect many small confirmations you might want to save all of them at the end in one go (?). Finally the payment method keeps track of what backend was used for this specific payment.

Parameters

- **`order`** – an `Order` instance
- **`amount`** – the paid amount
- **`transaction_id`** – the backend-specific transaction identifier
- **`save`** – a `bool` that indicates if the changes should be committed to the database.

2.3.2 Backend interface

The payment backend should define the following interface for the shop to be able do to anything sensible with it:

Attributes

`PaymentBackend.`**`backend_name`**

The name of the backend (to be displayed to users)

`PaymentBackend.`**`url_namespace`**

"slug" to prepend to this backend's URLs (acting as a namespace)

Methods

`PaymentBackend.`**`__init__`**`(`*shop*`)`

must accept a "shop" argument (to let the shop system inject a reference to it)

Parameters **`shop`** – an instance of the shop

`PaymentBackend.`**`get_urls`**`()`

should return a list of URLs (similar to urlpatterns), to be added to the URL resolver when urls are loaded. These will be namespaced with the url_namespace attribute by the shop system, so it shouldn't be done manually.

Security

In order to make your payment backend compatible with the `SHOP_FORCE_LOGIN` setting please make sure to add the `@shop_login_required` decorator to any views that your backend provides. See *How to secure your views* for more information.

2.4 How to create a shipping backend

- Shipping backends must be listed in settings.SHOP_SHIPPING_BACKENDS

2.4.1 Shop interface

While we could solve this by defining a superclass for all shipping backends, the better approach to plugins is to implement inversion-of-control, and let the backends hold a reference to the shop instead.

The reference interface for shipping backends is located at

class `shop.shipping.api.`**`ShippingAPI`**

2.4.2 Backend interface

The shipping backend should define the following interface for the shop to be able do to anything sensible with it:

Attributes

`backend_name`
The name of the backend (to be displayed to users)

`url_namespace`
"slug" to prepend to this backend's URLs (acting as a namespace)

Methods

`__init__`(*shop*)
must accept a "shop" argument (to let the shop system inject a reference to it)

Parameters **`shop`** – an instance of the shop

`get_urls`()
should return a list of URLs (similar to urlpatterns), to be added to the URL resolver when urls are loaded. These will be namespaced with the *`url_namespace`* attribute by the shop system, so it shouldn't be done manually.

Security

In order to make your shipping backend compatible with the `SHOP_FORCE_LOGIN` setting please make sure to add the `@shop_login_required` decorator to any views that your backend provides. See *How to secure your views* for more information.

2.5 How to interact with the cart

Interacting with the cart is probably the single most important thing shop implementers will want to do: e.g. adding products to it, changing quantities, etc...

There are roughly two different ways to interact with the cart: through Ajax, or with more standard post-and-refresh behavior.

2.5.1 Updating the whole cart

The normal form POST method is pretty straightforward - you simply POST to the cart's update URL (shop/cart/update/ by default) and pass it parameters as: update_item-<item id>=<new quantity> Items corresponding to the ID will be updated with the new quantity

2.5.2 Emptying the cart

Posting to shop/cart/delete empties the cart (the cart object is the same, but all cartitems are removed from it)

2.6 How to secure your views

Chances are that you don't want to allow your users to browse all views of the shop as anonymous users. If you set `SHOP_FORCE_LOGIN` to `True`, your users will need to login before proceeding to checkout.

When you add your own shipping and payment backends you will want to add this security mechanism as well. The problem is that the well known `@login_required` decorator will not work on class based views and it will also not work on functions that are members of a class.

For your convenience we provide three utilities that will help you to secure your views:

2.6.1 @on_method decorator

This decorator can be wrapped around any other decorator. It should be used on functions that are members of classes and will ignore the first parameter `self` and regard the second parameter as the first instead. More information can be found here.

Usage:

```
from shop.util.decorators import on_method, shop_login_required

class PayOnDeliveryBackend(object):

    backend_name = "Pay On Delivery"
    url_namespace = "pay-on-delivery"

    [...]

    @on_method(shop_login_required)
    def simple_view(self, request):
        [...]
```

2.6.2 @shop_login_required decorator

This decorator does the same as Django's @login_required decorator . The only difference is that it checks for the `SHOP_FORCE_LOGIN` setting. If that setting is `False`, login will not be required.

2.6.3 LoginMixin class

If you are using class based views for anything related to the shop you can use `shop.util.login_mixin.LoginMixin` to secure your views. More information on this can be found here. We are using a slightly modified version of that `LoginMixin` that makes sure to check for the `SHOP_FORCE_LOGIN` setting.

Usage:

```
class CheckoutSelectionView(LoginMixin, ShopTemplateView):
    template_name = 'shop/checkout/selection.html'

    [...]
```

CHAPTER 3

Advanced how to

More focused short articles, focusing on less general and more advanced use cases.

3.1 How to extend django SHOP models

(Instead of the default ones)

Some people might feel like the django SHOP models are not suitable for their project, or want to extend functionality for their specific needs.

This is a rather advanced use case, and most developers should hopefully be happy with the default models. It is however relatively easy to do.

All models you can override have a corresponding setting, which should contain the class path to the model you wish to use in its stead.

Note: While your models will be used, they will still be "called" by their default django SHOP name.

3.1.1 Example

Extending the Product model in django SHOP works like this:

```
# In myproject.models
from shop.models_bases import BaseProduct
class MyProduct(BaseProduct):
    def extra_method(self):
        return 'Yay'

    class Meta:
        pass

# In your project's settings.py, add the following line:
SHOP_PRODUCT_MODEL = 'myproject.models.MyProduct'
```

Important: Your model replacement must define a `Meta` class. Otherwise, it will inherit its parent's `Meta`, which will break things. The `Meta` class does not need to do anything important - it just has to be there.

Note: The above example is intentionally *not* using the same module as the examples given earlier in this documentation (myproject versus myshop). If MyProduct and Book were defined in the same module a circular import will result culminating in an error similar to: `django.core.exceptions.ImproperlyConfigured: Error importing backend myshop.models: "cannot import name Product". Check your SHOP_PRODUCT_MODEL setting.`

From a django interactive shell, you should now be able to do:

```
>>> from shop.models import Product
>>> p = Product.objects.all()[0] # I assume there is already at least one
>>> p.extra_method()
Yay
>>> p.__class__
<class object's class>
```

3.1.2 Settings

All available settings to control model overrides are defined in General Settings

3.2 How to use your own addressmodel

(Instead of the default one)

Some people might feel like the current addressmodel is not suitable for their project. You might be using a "client" + address model from an external application or simply want to write your own.

This is a rather advanced use case, and most developers should hopefully be happy with the default model. It is however relatively easy to do.

3.2.1 Deactivate the default addressmodel

Simple enough: just remove or comment the corresponding entry in your project's INSTALLED_APPS:

```
INSTALLED_APPS = (
...
'shop', # The django SHOP
#'shop.addressmodel', # <-- Comment this out
...
)
```

3.2.2 Hook your own model to the shop

To achieve this, simply add a *SHOP_ADDRESS_MODEL* to your settings file, and give the full python path to your Address model as a value:

```
SHOP_ADDRESS_MODEL = 'myproject.somepackage.MyAddress'
```

Your custom model *must* unfortunately have the following two fields defined for the checkout views to work:

```
user_shipping = models.OneToOneField(User, related_name='shipping_address', blank=True, null=True)
user_billing = models.OneToOneField(User, related_name='billing_address', blank=True, null=True)
```

This is to ensure that the views take handle “attaching” the address objects to the User (or the session if the shopper is a guest).

We recommend adding the `as_text()` method to your address model. This ‘collects’ all fields and returns them in one string. This string will be saved to the order (to `billing_address_text` or `shipping_address_text` accordingly) during checkout view.

You are obviously free to subclass these views and hook in your own behavior.

CHAPTER 4

Reference

Reference sheets and lists regarding django SHOP

4.1 Plugins

Django SHOP defines 3 types of different plugins for the time being:

1. Cart modifiers
2. Shipping modules
3. Payment modules

4.1.1 Cart modifiers

Cart modifiers are plugins that modify the cart's contents.

Rough categories could be discount modules or tax modules: rules for these modules are invariant, and should be "stacked".

Example: "CDs are buy two get one free this month", "orders over $500 get a 10% discount"

How they work

Cart modifiers should extend the `shop.cart.cart_modifiers_base.BaseCartModifier` class.

Users must register these filters in the settings.SHOP_PRICE_MODIFIERS settings entry. Modifiers will be iterated and function in the same fashion as django middleware classes.

`BaseCartModifier` defines a set of methods that implementations should override, and that are called for each cart item/cart when the cart's `update()` method is called.

Example implementations

You can refer to the `shop.cart.modifiers` package to see some example implementations

4.1.2 Shipping backends

Shipping backends differ from price modifiers in that there must be only one shipping backend selected per order (the shopper must choose which delivery method to use)

Shipping costs should be calculated on an `Order` object, not on a `Cart` object (`Order` instances are fully serialized in the database for archiving purposes).

How they work

Shipping backends need to be registered in the SHOP_SHIPPING_BACKENDS Django setting. They do not need to extend any particular class, but need to expose a specific interface, as defined in *Backend interface*.

The core functionality the shop exposes is the ability to retrieve the current `Order` object (and all it's related bits and pieces such as extra price fields, line items etc...) via a convenient `self.shop.get_order()` call. This allows for your module to be reused relatively easily should another shop system implement this interface.

On their part, shipping backends should expose at least a *`get_urls()`* method, returning a `urlpattern`-style list or urls. This allows backend writers to have almost full control of the shipping process (they can create views and make them available to the URL resolver).

Please note that module URLs should be namespaced, and will be added to the `ship/` URL namespace. This is a hard limitation to avoid URL name clashes.

4.1.3 Payment backends

Payment backends must also be selected by the end user (the shopper). Theses modules take care of the actual payment processing.

How they work

Similar to shipping backends, payment backends do not need to extend any particular class, but need to expose a specific interface, as defined in *Backend interface*.

They also obtain a reference to the shop, with some convenient methods defined such as `self.shop.get_order()`.

They must also define a *`get_urls()`* method, and all defined URLs will be namespaced to `pay/`.

4.2 General Settings

This is a small list of the settings defined in django SHOP.

4.2.1 SHOP_PAYMENT_BACKENDS

A list (or iterable) of payment backend class paths. These classes will be used as the active payment backends by the checkout system, and so anything in this list will be shown to the customer for him/her to make a decision

4.2.2 SHOP_SHIPPING_BACKENDS

In a similar fashion, this must be a list of shipping backends. This list is used to display to the end customer what shipping options are available to him/her during the checkout process.

4.2.3 SHOP_CART_MODIFIERS

These modifiers function like the django middlewares. The cart will call each of these classes, in order, every time it is displayed. They are passed every item in the cart, as well as the cart itself.

4.2.4 SHOP_FORCE_LOGIN

If `True`, all views after the `CartDetails` view will need the user to be authenticated. An anonymous user will be redirected to your login url. Please read more on authentication in Django's official authentication documentation . By default it's set to `False`.

4.2.5 SHOP_PRICE_FORMAT

Used by the *priceformat* template filter to format the price. Default is `'%0.2f'`

4.3 Backend specific Settings

Some backends define extra settings to tweak their behavior. This should be an exhaustive list of all of the backends and modifiers included in the trunk of django SHOP.

4.3.1 SHOP_SHIPPING_FLAT_RATE

(Optional) The "flat rate" shipping module uses this to know how much to charge. This should be a string, and will be converted to a Decimal by the backend.

4.4 Extensibility Settings

Theses settings allow developers to extend the shop's functionality by replacing models with their own models. More information on how to use these settings can be found in the How to extend django SHOP models section.

4.4.1 SHOP_CART_MODEL

(Optional) A python classpath to the class you want to replace the Cart model with. Example value: *myproject.models.MyCartModel*

4.4.2 SHOP_ADDRESS_MODEL

(Optional) A python classpath to the class you want to replace the `shop.addressmodel.models.Address` model with. See How to use your own addressmodel for a more complete example.

Example value: *myproject.models.MyAddressModel*

4.4.3 SHOP_ORDER_MODEL

(Optional) A python classpath to the class you want to replace the `shop.models.Order` model with.

Example value: *myproject.models.MyOrderModel*

CHAPTER 5

The name

The official name of this project is **django SHOP**.

Django SHOP should be capitalised at the start of sentences and in title-case headings.

When referring to the package, repositories and any other things in which spaces are not permitted, use **django-shop**.

s

shop.order_signals, 6
shop.payment.api, 13

Symbols

__init__(), 16
__init__() (shop.payment.api.PaymentBackend method), 15

A

active (shop.models.Product attribute), 11
add_extra_info() (shop.payment.api.PaymentAPI method), 14

B

backend_name, 16
backend_name (shop.payment.api.PaymentBackend attribute), 15

C

confirm_payment() (shop.payment.api.PaymentAPI method), 15

D

date_added (shop.models.Product attribute), 11

G

get_order() (shop.payment.api.PaymentAPI method), 14
get_order_for_id() (shop.payment.api.PaymentAPI method), 14
get_order_short_name() (shop.payment.api.PaymentAPI method), 14
get_order_subtotal() (shop.payment.api.PaymentAPI method), 14
get_order_total() (shop.payment.api.PaymentAPI method), 14
get_order_unique_id() (shop.payment.api.PaymentAPI method), 14
get_urls(), 16
get_urls() (shop.payment.api.PaymentBackend method), 15

I

is_order_complete() (shop.payment.api.PaymentAPI method), 14
is_order_payed() (shop.payment.api.PaymentAPI method), 14

L

last_modified (shop.models.Product attribute), 11

N

name (shop.models.Product attribute), 11

P

PaymentAPI (class in shop.payment.api), 13
Python Enhancement Proposals
    PEP 8, 9

S

shop.models.Product (built-in class), 11
shop.order_signals (module), 6
shop.order_signals.cancelled (built-in variable), 7
shop.order_signals.completed (built-in variable), 7
shop.order_signals.confirmed (built-in variable), 7
shop.order_signals.payment_selection (built-in variable), 6
shop.order_signals.processing (built-in variable), 6
shop.order_signals.shipped (built-in variable), 7
shop.payment.api (module), 13
shop.shipping.api.ShippingAPI (built-in class), 16
slug (shop.models.Product attribute), 11

U

unit_price (shop.models.Product attribute), 11
url_namespace, 16
url_namespace (shop.payment.api.PaymentBackend attribute), 15