django-oscar Documentation

Release 0.3

David Winterbottom

June 19, 2014

Contents

1 Take a peek **3**
1.1 Browse the sandbox site . . . 3
1.2 Running the sandbox locally . . . 3
1.3 Browse some real Oscar implementations . . . 4

2 Start building your own shop **5**
2.1 Install by hand . . . 5
2.2 Install using Tangent's boilerplate django project . . . 7
2.3 Next steps . . . 8

3 Building an e-commerce site: the key questions **9**
3.1 Catalogue . . . 9
3.2 Pricing, stock and availability . . . 10
3.3 Shipping . . . 10
3.4 Payment . . . 11

4 Recipes **13**
4.1 Customisation . . . 13
4.2 Catalogue . . . 18
4.3 Pricing, stock and availability . . . 18
4.4 Tax . . . 19
4.5 Shipping . . . 20

5 Getting help **23**
5.1 Solutions to common problems . . . 23

6 Design decisions **25**
6.1 Core models are abstract . . . 25
6.2 Classes are loaded dynamically . . . 25
6.3 All views are class-based . . . 26
6.4 Templates can be overridden . . . 26

7 Reference **27**
7.1 Oscar specific settings . . . 27
7.2 Signals . . . 28

8 Contributing **31**
8.1 Installation . . . 31
8.2 Writing docs . . . 31

8.3 Conventions 31

9 Indices and tables **33**

Oscar is an e-commerce framework for Django designed for building domain-driven applications. It is structured so that the core business objects can be customised to suit the domain at hand. In this way, your application can accurately model its domain, making feature development and maintenance much easier.

Features:

- Any product type can be handled, including downloadable products, subscriptions, variant products (eg a T-shirt in different sizes and colours).
- Customisable products, such as T-shirts with personalised messages.
- Can be used for large catalogues - Oscar is used in production by sites with more than 20 million products.
- Multiple fulfillment partners for the same product.
- Range of merchandising blocks for promoting products throughuout your site.
- Sophisticated offers that support virtually any kind of offer you can think of - multibuys, bundles, buy X get 50% of Y etc
- Vouchers
- Comprehensive dashboard
- Support for split payment orders
- Extension libraries available for PayPal, GoCardless, DataCash and more

Oscar is a good choice if your domain has non-trivial business logic. Oscar's flexibility means it's straightforward to implement business rules that would be difficult to apply in other frameworks.

Example requirements that Oscar applications already handle:

- Paying for an order with multiple payment sources (eg using a bankcard, voucher, giftcard and business account).
- Complex access control rules governing who can view and order what.
- Supporting a hierarchy of customers, sales reps and sales directors - each being able to "masquerade" as their subordinate users.
- Multi-lingual products and categories

Oscar is developed by Tangent Labs, a London-based digital agency. It is used in production in several applications to sell everything from beer mats to ipads. The source is on Github - contributions welcome.

CHAPTER 1

Take a peek

There are several ways to get a feel for Oscar and what it can do.

1.1 Browse the sandbox site

There is a demo Oscar site, built hourly from HEAD of the master branch (unstable): http://sandbox.oscar.tangentlabs.co.uk

It is intended to be a vanilla install of Oscar, using the default templates and styles. This is the blank canvas upon which you an build your application.

It only has two customisations on top of Oscar's core:

- Two shipping methods are specified so that the shipping method step of checkout is not skipped. If there is only one shipping method (which is true of core oscar) then the shipping method step is skipped as there is no choice to be made.
- A profile class is specified which defines a few simple fields. This is to demonstrate the account section of Oscar, which introspects the profile class to build a combined User and Profile form.

1.2 Running the sandbox locally

It's pretty straightforward to get the sandbox site running locally so you can play around with the sourcecode.

Install Oscar and its dependencies within a virtualenv:

```
git clone git://github.com/tangentlabs/django-oscar.git
cd django-oscar
mkvirtualenv oscar
make contribute
```

This will install all dependencies required for developing Oscar and create a simple database populated with products.

You can then browse a sample Oscar site using Django's development server:

```
cd sandbox
./manage.py runserver
```

Note that some things are deliberately not implemented within core Oscar as they are domain-specific. For instance:

- All tax is set to zero
- The two shipping methods are both free

- No payment is required to submit an order

1.2.1 I've found a problem!

Good for you - please report them in Github's issue tracker.

1.3 Browse some real Oscar implementations

There are several Oscar implementations in production, although several are B2B and hence can't be browsed by the public. Here's a few public ones to have a look at:

- http://www.landmarkonthenet.com - Landmark is part of Tata Group. This site has a catalogue size of more than 20 million products and integrates with many partners such as Gardners, Ingram, Nielsen, Citibank, Qwikcilver and SAP.
- http://www.dolbeau.ca/ - "Dolbeau delivers weekly limited editions of handcrafted luxury menswear"

CHAPTER 2

Start building your own shop

For simplicity, let's assume you're building a new e-commerce project from scratch and have decided to use Oscar. Let's call this shop 'frobshop'

Tip: You can always review the set-up of the Sandbox site in case you have trouble with the below instructions.

2.1 Install by hand

Install Oscar (which will install Django as a dependency), then create the project:

```
pip install django-oscar
django-admin.py startproject frobshop
```

This will create a folder `frobshop` for your project.

2.1.1 Settings

Now edit your settings file `frobshop.frobshop.settings.py` to specify a database (we use SQLite for simplicity):

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

then add `oscar.apps.basket.middleware.BasketMiddleware` to `MIDDLEWARE_CLASSES`, and set `TEMPLATE_CONTEXT_PROCESSORS` to:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.request",
    "django.core.context_processors.debug",
    "django.core.context_processors.i18n",
    "django.core.context_processors.media",
    "django.core.context_processors.static",
    "django.contrib.messages.context_processors.messages",
    'oscar.apps.search.context_processors.search_form',
    'oscar.apps.promotions.context_processors.promotions',
    'oscar.apps.checkout.context_processors.checkout',
    'oscar.core.context_processors.metadata',
)
```

Next, modify `INSTALLED_APPS` to be a list, add `South` and append Oscar's core apps:

```
from oscar import get_core_apps
INSTALLED_APPS = [
    'django.contrib.auth',
    ...
    'south',
] + get_core_apps()
```

and set your auth backends to:

```
AUTHENTICATION_BACKENDS = (
    'oscar.apps.customer.auth_backends.Emailbackend',
    'django.contrib.auth.backends.ModelBackend',
)
```

to allow customers to sign in using an email address rather than a username.

Modify your `TEMPLATE_DIRS` to include the main Oscar template directory:

```
from oscar import OSCAR_MAIN_TEMPLATE_DIR
TEMPLATE_DIRS = TEMPLATE_DIRS + (OSCAR_MAIN_TEMPLATE_DIR,)
```

Oscar currently uses Haystack for search so you need to specify:

```
HAYSTACK_SITECONF = 'oscar.search_sites'
HAYSTACK_SEARCH_ENGINE = 'dummy'
```

When moving towards production, you'll obviously need to switch to a real search backend.

The last addition to the settings file is to import all of Oscar's default settings:

```
from oscar.defaults import *
```

2.1.2 URLs

Alter your `frobshop/urls.py` to include Oscar's URLs:

```
from django.conf.urls import patterns, include, url
from oscar.app import shop

urlpatterns = ('',
    (r'', include(shop.urls))
)
```

2.1.3 Database

Then create the database and the shop should be browsable:

```
python manage.py syncdb --noinput
python manage.py migrate
```

You should now have a running Oscar install that you can browse.

2.2 Install using Tangent's boilerplate django project

The easiest way to get started is to use Tangent's template django project although it is tailored to an agency structure which may not suit everyone.

Set up a virtualenv, and create a new project using the `startproject` management command:

```
mkvirtualenv frobshop
pip install Django
django-admin.py startproject frobshop \
    --template=https://github.com/tangentlabs/tangent-django-boilerplate/zipball/master
```

This will create a folder `frobshop` which is an entire templated project that follows Tangent's conventions. The structure is:

```
frobshop/
    docs/
    www/
        conf/
        deploy/
        public/
        static/
        templates/
        manage.py
        settings.py
        settings_test.py
        urls.py
        urls_oscar.py
    README.rst
    fabconfig.py
    fabfile.py
    deploy-to-test.sh
    deploy-to-stage.sh
    deploy-to-prod.sh
```

Replace a few files with Oscar-specific versions (the templated project can be used for non-Oscar projects too):

```
mv frobshop/www/urls{_oscar,}.py
mv frobshop/www/deploy/requirements{_oscar,}.py
mv frobshop/www/conf/default{_oscar,}.py
```

Install dependencies:

```
cd frobshop/www
pip install -r deploy/requirements.txt
```

Create database:

```
python manage.py syncdb --noinput
python manage.py migrate
```

And that should be it.

2.3 Next steps

The next step is to implement the business logic of your domain on top of Oscar. The fun part.

CHAPTER 3

Building an e-commerce site: the key questions

When building an e-commerce site, there are several components whose implementation is strongly domain-specific. That is, every site will have different requirements for how such a component should operate. As such, these components cannot easily be modelled using a generic system - no configurable system will be able to accurately capture all the domain-specific behaviour required.

The design philosophy of Oscar is to not make a decision for you here, but to provide the environment where any domain logic can be implemented, no matter how complex.

This document lists the components which will require implementation according to the domain at hand. These are the key questions to answer when building your application. Much of Oscar's documentation is in the form of "recipes" that explain how to solve the questions listed here. Each question links to the relevant recipes.

3.1 Catalogue

3.1.1 What are your product types?

Are you selling books, DVDs, clothing, downloads or fruit and veg? You will need to capture the attributes of your product types within your models. Oscar divides products into 'product classes' which each have their own set of attributes.

- *How to model your catalogue*
- *Importing a catalogue*

3.1.2 How is your catalogue organised?

How are products organised within the site? A common pattern is to have a single category tree where each product belongs to one category which sits within a tree structure of other categories. However, there are lots of other options such as having several separate taxonomy trees (eg split by brand, by theme, by product type). Other questions to consider:

- Can a product belong to more than one category?
- Can a category sit in more than one place within the tree? (eg a "children's fiction" category might sit beneath "children's books" and "fiction").
- *How to customise an app*
- *How to customise models*
- *How to override a core class*

3.1.3 How are products managed?

Is the catalogue managed by a admin using a dashboard, or though an automated process, such as processing feeds from a fulfillment system? Where are your product images going to be served from?

- *How to disable an app's URLs*

3.2 Pricing, stock and availability

3.2.1 How is tax calculated?

3.2.2 What availability messages are shown to customers?

Based on the stock information from a fulfilment partner, what messaging should be displayed on the site?

- *How to configure stock messaging*

3.2.3 Do you allow pre- and back-orders

An pre-order is where you allow a product to be bought before it has been published, while a back-order is where you allow a product to be bought that is currently out of stock.

3.3 Shipping

3.3.1 How are shipping charges calculated?

There are lots of options and variations here. Shipping methods and their associated charges can take a variety of forms, including:

- A charge based on the weight of the basket
- Charging a pre-order and pre-item charge
- Having free shipping for orders above a given threshold

Recipes:

- *How to configure shipping*

3.3.2 Which shipping methods are available?

There's often also an issue of which shipping methods are available, as this can depend on:

- The shipping address (eg overseas orders have higher charges)
- The contents of the basket (eg free shipping for downloadable products)
- Who the user is (eg sales reps get free shipping)

Oscar provides classes for free shipping, fixed charge shipping, pre-order and per-product item charges and weight-based charges. It is provides a mechanism for determing which shipping methods are available to the user.

Recipes:

- *How to configure shipping*

3.4 Payment

3.4.1 How are customers going to pay for orders?

Often a shop will have a single mechanism for taking payment, such as integrating with a payment gateway or using PayPal. However more complicated projects will allow users to combine several different payment sources such as bankcards, business accounts and giftcards.

Possible payment sources include:

- Bankcard
- Google checkout
- PayPal
- Business account
- Managed budget
- Giftcard
- No upfront payment but send invoices later

The checkout app within django-oscar is suitable flexible that all of these methods (and in any combination) is supported. However, you will need to implement the logic for your domain by subclassing the relevant view/util classes.

Domain logic is often required to:

- Determine which payment methods are available to an order;
- Determine if payment can be split across sources and in which combinations;
- Determine the order in which to take payment
- Determine how to handle failing payments (this can get complicated when using multiple payment sources to pay for an order).
- *How to configure shipping*

3.4.2 When will payment be taken?

A common pattern is to 'pre-auth' a bankcard at the point of checkout then 'settle' for the appropriate amouts when the items actually ship. However, sometimes payment is taken up front. Often you won't have a choice due to limitations of the payment partner you need to integrate with.

- Will the customer be debited at point of checkout, or when the items are dispatched?
- If charging after checkout, when are shipping charges collected?
- What happens if an order is cancelled after partial payment?

CHAPTER 4

Recipes

Recipes are simple guides to solving common problems that occur when creating e-commerce projects.

4.1 Customisation

4.1.1 How to customise an app

A core part of how oscar can be customised is to create a local version of one of oscar's apps so that it can be modified and extended. Creating a local version of an app allows customisation of any of the classes within the corresponding app in oscar.

The way this is done involves a few steps, which are detailed here.

Method

1. Create an app within your project with the same "app label" as an app in oscar. Eg, to create a local version of `oscar.apps.order`, create something like `myproject.order`.
2. Ensure the `models.py` in your local app imports all the models from oscar's version:

   ```
   # models.py
   ```

   from oscar.apps.order.models import *
3. Replace oscar's version of the app with your new version in `INSTALLED_APPS`.

Worked example

Suppose you want to modify the homepage view class, which by default is defined in `oscar.apps.promotions.views.HomeView`. This view is bound to a URL within the `PromotionsApplication` class in `oscar.apps.promotions.app` - hence we need to override this application class to be able to use a different view.

By default, your base `urls.py` should include oscar's URLs as so:

```
# urls.py
from oscar.app import application

urlpatterns = patterns('',
    ...
```

```
    (r'', include(application.urls)),
)
```

To get control over the mapping between URLs and views, you need to use a local `application` instance, that (optionally) subclasses oscar's. Hence, create `myproject/app.py` with contents:

```
# myproject/app.py
from oscar.app import Shop

class BaseApplication(Shop):
    pass

application = BaseApplication()
```

No customisation for now, that will come later, but you now have control over which URLs and view functions are used.

Now hook this up in your `urls.py`:

```
# urls.py
from myproject.app import application

urlpatterns = patterns('',
    ...
    (r'', include(application.urls)),
)
```

The next step is to create a local app with the same name as the app you want to override:

```
mkdir myproject/promotions
touch myproject/promotions/__init__.py
touch myproject/promotions/models.py
```

The `models.py` file should import all models from the oscar app being overridden:

```
# myproject/promotions/models.py
from oscar.apps.promotions.models import *
```

Now replace `oscar.apps.promotions` with `myproject.promotions` in the `INSTALLED_APPS` setting in your settings file.

Now create a new homepage view class in `myproject.promotions.views` - you can subclass oscar's view if you like:

```
from oscar.apps.promotions.views import HomeView as CoreHomeView

class HomeView(CoreHomeView):
    template_name = 'promotions/new-homeview.html'
```

In this example, we set a new template location but it's possible to customise the view in any imaginable way.

Next, create a new `app.py` for your local promotions app which maps your new `HomeView` class to the homepage URL:

```
# myproject/promotions/app.py
from oscar.apps.promotions import PromotionsApplication as CorePromotionsApplication

from myproject.promotions.views import HomeView

class PromotionsApplication(CorePromotionsApplication):
    home_view  = HomeView
```

```
application = PromotionsApplication()
```

Finally, hook up the new view to the homepage URL:

```
# myproject/app.py
from oscar.app import Shop

from myproject.promotions.app import application as promotions_app

class BaseApplication(Shop):
    promotions_app = promotions_app
```

Quite long-winded, but once this step is done, you have lots of freedom to customise the app in question.

Other points of note

One pain point with replacing one of oscar's apps with a local one in `INSTALLED_APPS` is that template tags are lost from the original app and need to be manually imported. This can be done by creating a local version of the template tags files:

```
mkdir myproject/templatetags
```

and importing the tags from oscar's corresponding file:

```
# myproject/promotions/templatetags/promotion_tags.py
from oscar.apps.promotions.templatetags.promotion_tags import *
```

This isn't great but we haven't found a better way as of yet.

4.1.2 How to customise models

You must first create a local version of the app that you wish to customise. This involves creating a local app with the same name and importing the equivalent models from oscar into it.

Example

Suppose you want to add a video_url field to the core product model. This means that you want your application to use a subclass of `oscar.apps.catalogue.models.Product` which has an additional field.

The first step is to create a local version of the "catalogue" app. At a minimum, this involves creating `catalogue/models.py` within your project and changing `INSTALLED_APPS` to point to your local version rather than oscar's.

Next, you can modify the `Product` model through subclassing:

```
# yourproject/catalogue/models.py

from oscar.apps.catalogue.abstract_models import AbstractProduct

class Product(AbstractProduct):
    video_url = models.URLField()
```

Now, running `./manage.py syncdb` will create the product model with your additional field

4.1.3 How to override a core class

Example

Suppose you want to alter the way order number's are generated. By default, the class `oscar.apps.order.utils.OrderNumberGenerator` is used. To change the behaviour, you need to ensure that you have a local version of the `order` app (ie `INSTALLED_APPS` should contain `yourproject.order`, not `oscar.apps.order`). Then create a class within your `order` app which matches the module path from oscar: `order.utils.OrderNumberGenerator`. This could subclass the class from oscar or not. An example implementation is:

```
# yourproject/order/utils.py

from oscar.apps.order.utils import OrderNumberGenerator as CoreOrderNumberGenerator


class OrderNumberGenerator(CoreOrderNumberGenerator):

    def order_number(self, basket=None):
        num = super(OrderNumberGenerator, self).order_number(basket)
        return "SHOP-%s" % num
```

the same module path as the one from oscar, that is,

Discussion

This principle of overriding classes from modules is an important feature of oscar and makes it easy to customise virtually any functionality from the core. For this to work, you must ensure that:

1. You have a local version of the app, rather than using oscar's directly
2. Your local class has the same module path relative to the app as the oscar class being overridden

4.1.4 How to customise templates

Assuming you want to use oscar's templates in your project, there are two options. You don't have to though - you could write all your own templates if you like. If you do this, it's probably best to start with a straight copy of all of oscar's templates so you know all the files that you need to re-implement.

Anyway - here are the two options for customising.

Method 1 - Forking

One option is always just to fork the template into your local project so that it comes first in the include path.

Say you want to customise `base.html`. First you need a project-specific templates directory that comes first in the include path. You can set this up as so:

```
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
)

import os
location = lambda x: os.path.join(os.path.dirname(os.path.realpath(__file__)), '..', x)
TEMPLATE_DIRS = (
    location('templates'),
)
```

Next copy oscar's `base.html` into your templates directory and customise it to suit your needs.

The downsides of this method are that it involves duplicating the file from oscar in a way that breaks the link with upstream. Hence, changes to oscar's `base.html` won't be picked up by your project as you will have your own version.

Method 2 - Subclass parent but use same template path

There is a trick you can perform whereby oscar's templates can be accessed via two paths. This is outlined in the django wiki.

This basically means you can have a `base.html` in your local templates folder that extends oscar's `base.html` but only customises the blocks that it needs to.

Oscar provides a helper variable to make this easy. First, set up your template configuration as so:

```
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
)

import os
location = lambda x: os.path.join(os.path.dirname(os.path.realpath(__file__)), '..', x)
from oscar import OSCAR_PARENT_TEMPLATE_DIR
TEMPLATE_DIRS = (
    location('templates'),
    OSCAR_PARENT_TEMPLATE_DIR,
)
```

The `OSCAR_PARENT_TEMPLATE_DIR` points to the directory above oscar's normal templates directory. This means that `path/to/oscar/template.html` can also be reached via `templates/path/to/oscar/template.html`.

Hence to customise `base.html`, you can have an implementation like:

```
# base.html
{% extends 'templates/base.html' %}

...
```

No real downsides to this one other than getting your front-end people to understand it.

4.1.5 How to disable an app's URLs

Suppose you don't want to use Oscar's dashboard but use your own. The way to do this is to modify the URLs config to exclude the URLs from the app in question.

You need to use your own root 'application' instance which gives you control over the URLs structure. So your root `urls.py` should have:

```
# urls.py
from myproject.app import application

urlpatterns = patterns('',
    ...
    (r'', include(application.urls)),
)
```

where `application` is a subclass of `oscar.app.Shop` which overrides the link to the dashboard app:

```
# myproject/app.py
from oscar.app import Shop
from oscar.core.application import Application

class MyShop(Shop):

    # Override the core dashboard_app instance to use a blank application
    # instance.  This means no dashboard URLs are included.
    dashboard_app = Application()
```

The only remaining task is to ensure your templates don't reference any dashboard URLs.

4.2 Catalogue

4.2.1 How to create categories

The simplest way is to use a string which represents the breadcrumbs:

```
from oscar.apps.catalogue.categories import create_from_breadcrumbs

categories = (
    'Food > Cheese',
    'Food > Meat',
    'Clothes > Man > Jackets',
    'Clothes > Womam > Skirts',
)
for breadcrumbs in categories:
    create_from_breadcrumbs(breadcrumbs)
```

4.2.2 How to model your catalogue

Related recipes:

- *How to customise an app*
- *How to customise models*

4.2.3 Importing a catalogue

...

4.3 Pricing, stock and availability

4.3.1 How to enforce stock rules

You can enfore stock validation rules using signals. You just need to register a listener to the BasketLine pre_save signal that checks the line is valid. For example:

```
@receiver(pre_save, sender=Line)
def handle_line_save(sender, **kwargs):
    if 'instance' in kwargs:
        quantity = int(kwargs['instance'].quantity)
        if quantity > 4:
            raise InvalidBasketLineError("You are only allowed to purchase a maximum of 4 of these")
```

4.3.2 How to configure stock messaging

Stock messaging is controlled on a per-partner basis. A product's stockrecord has the following methods for messaging:

Both these methods delegate to a "partner wrapper" instance. These are defined in the `OSCAR_PARTNER_WRAPPERS` setting which is a dict mapping from partner name to a class path, for instance:

```
# settings.py
OSCAR_PARTNER_WRAPPERS = {
    'Partner A': 'myproject.wrappers.PartnerAWrapper',
}
```

The default wrapper is `oscar.apps.partner.wrappers.DefaultWrapper`, which provides methods of the same name.

Custom wrappers should subclass this class and override the appropriate methods. Here's an example wrapper that provides custom availability messaging:

```
# myproject/wrappers.py
from oscar.apps.partner import wrappers


class PartnerAWrapper(wrappers.DefaultWrapper):

    def availability(self, stockrecord):
        if stockrecord.net_stock_level > 0:
            return "Available to buy now!"
        return "Sorry, not available"

    def availability_code(self, stockrecord):
        if stockrecord.net_stock_level > 0:
            return "icon_tick"
        return "icon_cross"
```

4.4 Tax

4.4.1 How to apply tax exemptions

Problem

The tax a customer pays depends on the shipping address of his/her order.

Solution

Use custom basket middleware to set the tax status of the basket.

The default Oscar basket middleware is:

```
'oscar.apps.basket.middleware.BasketMiddleware'
```

To alter the tax behaviour, replace this class with one within your own project that subclasses Oscar's and extends the `get_basket` method. For example, use something like:

```
from oscar.apps.basket.middleware import BasketMiddleware
from oscar.apps.checkout.utils import CheckoutSessionData

class MyBasketMiddleware(BasketMiddleware):

    def get_basket(self, request):
        basket = super(MyBasketMiddleware, self).get_basket(request)
        if self.is_tax_exempt(request):
            basket.set_as_tax_exempt()
        return basket

    def is_tax_exempt(self, request):
        country = self.get_shipping_address_country(request)
        if country is None:
            return False
        return country.iso_3166_1_a2 not in ('GB',)

    def get_shipping_address_country(self, request):
        session = CheckoutSessionData(request)
        if not session.is_shipping_address_set():
            return None
        addr_id = session.shipping_user_address_id()
        if addr_id:
            # User shipping to address from address book
            return UserAddress.objects.get(id=addr_id).country
        else:
            fields = session.new_shipping_address_fields()
```

Here we are using the checkout session wrapper to check if the user has set a shipping address. If they have, we extract the country and check its ISO 3166 code.

It is straightforward to extend this idea to apply custom tax exemptions to the basket based of different criteria.

4.5 Shipping

4.5.1 How to configure shipping

Checkout flow

Oscar's checkout is set-up to follow the following steps:

1. Manage basket
2. Enter/choose shipping address
3. Choose shipping method
4. Choose payment method
5. Preview
6. Enter payment details and submit

Determining the methods available to a user

At the shipping method stage, we use a repository object to look up the shipping methods available to the user. These methods typically depend on:

- the user in question (eg. staff get cheaper shipping rates)
- the basket (eg. shipping is charged based on the weight of the basket)
- the shipping address (eg. overseas shipping is more expensive)

The default repository is `oscar.apps.shipping.repository.Repository`, which has a method `get_shipping_methods` for returning all available methods. By default, the returned method will be `oscar.apps.shipping.methods.Free`.

Set a custom shipping methods

To apply your domain logic for shipping, you will need to override the default repository class (see *How to override a core class*) and alter the implementation of the `get_shipping_methods` method. This method should return a list of "shipping method" classes already instantiated and holding a reference to the basket instance.

Building a custom shipping method

A shipping method class must define two methods:

```
method.basket_charge_incl_tax()
method.basket_charge_excl_tax()
```

whose responsibilities should be clear. You can subclass `oscar.apps.shipping.base.ShippingMethod` to provide the basic functionality.

Built-in shipping methods

Oscar comes with several built-in shipping methods which are easy to use with a custom repository.

- `oscar.apps.shipping.methods.Free`. No shipping charges.
- `oscar.apps.shipping.methods.WeightBased`. This is a model-driven method that uses two models: `WeightBased` and `WeightBand` to provide charges for different weight bands. By default, the method will calculate the weight of a product by looking for a 'weight' attribute although this can be configured.
- `oscar.apps.shipping.methods.FixedPrice`. This simply charges a fixed price for shipping, irrespective of the basket contents.
- `oscar.apps.shipping.methods.OrderAndItemCharges`. This is a model which specifies a per-order and a per-item level charge.

CHAPTER 5

Getting help

If you're stuck with a problem, try checking the Google Groups archive to see if someone has encountered it before. If not, then try asking on the mailing list django-oscar@googlegroups.com. If it's a common question, then we'll write up the solution as a recipe.

5.1 Solutions to common problems

5.1.1 My model customisations aren't picked up

Symptom

You're trying to customise one of Oscar's models but your new fields don't seem to get picked up. For example, you are trying to add a field to the product model

Cause

Oscar's models are being imported before your customised one. Due to the way model registration works with Django, the order in which models are imported is important.

Somewhere in your codebase is an import from `oscar.apps.*.models` that is being executed before your models are parsed.

Solution

Find and remove the import statement that is importing Oscar's models.

In your overriding `models.py`, ensure that you import Oscar's models *after* your custom ones have been defined.

If other modules need to import these models then import from your local module, not from Oscar directly

Mailing list threads

https://groups.google.com/forum/?fromgroups#!topic/django-oscar/oMdAB7zJLlI

If you find a bug, please report it in the Github issue tracker

CHAPTER 6

Design decisions

The central aim of oscar is to provide a solid core of an e-commerce project that can be extended and customised to suit the domain at hand. This is acheived in several ways:

6.1 Core models are abstract

Online shops can vary wildly, selling everything from turnips to concert tickets. Trying to define a set of Django models capable for modelling all such scenarios is impossible - customisation is what matters.

One way to model your domain is to have enormous models that have fields for every possible variation; however, this is unwieldy and ugly.

Another is to use the Entity-Attribute-Value pattern to use add meta-data for each of your models. However this is again ugly and mixes meta-data and data in your database (it's an SQL anti-pattern).

Oscar's approach to this problem is to have have minimal but abstract models where all the fields are meaningful within any e-commerce domain. Oscar then provides a mechanism for subclassing these models within your application so domain-specific fields can be added.

Specifically, in many of oscar's apps, there is an `abstract_models.py` module which defines these abstract classes. There is also an accompanying `models.py` which provides an empty but concrete implementation of each abstract model.

6.2 Classes are loaded dynamically

To enable sub-apps to be overridden, oscar classes are loading generically using a special `get_class` function. This looks at the `INSTALLED_APPS` tuple to determine the appropriate app to load a class from.

Sample usage:

```
from oscar.core.loading import get_class

Repository = get_class('shipping.repository', 'Repository')
```

This is a replacement for the usual:

```
from oscar.apps.shipping.repository import Repository
```

It is effectively an extension of Django's `django.db.models.get_model` function to work with arbitrary classes.

The `get_class` function looks through your `INSTALLED_APPS` for a matching module to the one specified and will load the classes from there. If the matching module is not from oscar's core, then it will also fall back to the equivalent module if the class cannot be found.

This structure enables a project to create a local `shipping.repository` module and subclass and extend the classes from `oscar.app.shipping.repository`. When Oscar tries to load the `Repository` class, it will load the one from your local project.

6.3 All views are class-based

This enables any view to be subclassed and extended within your project.

6.4 Templates can be overridden

This is a common technique relying on the fact that the template loader can be configured to look in your project first for templates, before it uses the defaults from oscar.

CHAPTER 7

Reference

Contents:

7.1 Oscar specific settings

Oscar provides a number of configurable settings used to confugre the system.

- Available settings
- Deprecated settings

7.1.1 Available settings

OSCAR_DEFAULT_CURRENCY

Default: `None` (This is a required field)

This should be the symbol of the currency you wish Oscar to use by default.

OSCAR_BASKET_COOKIE_LIFETIME

Default: 604800 (1 week in seconds)

The time to live for the basket cookie in seconds

OSCAR_IMAGE_FOLDER

Default: `images/products/%Y/%m/`

The path for uploading images to.

OSCAR_RECENTLY_VIEWED_PRODUCTS

Default: 4

The number of recently viewed products to store

OSCAR_SEARCH_SUGGEST_LIMIT

Default: 10

The number of suggestions that the search 'suggest' function should return at maximum

7.1.2 Deprecated settings

There are currently no deprecated settings in oscar

7.2 Signals

Oscar implements a number of custom signals that provide useful hook-points for adding functionality.

7.2.1 product_viewed

Raised when a product detail page is viewed.

Arguments sent with this signal:

`product` The product being viewed

`user` The user in question

`request` The request instance

`response` The response instance

7.2.2 product_search

Raised when a search is performed.

Arguments sent with this signal:

`query` The search term

`user` The user in question

7.2.3 basket_addition

Raised when a product is added to a basket

Arguments sent with this signal:

`product` The product being added

`user` The user in question

7.2.4 voucher_addition

Raised when a valid voucher is added to a basket

Arguments sent with this signal:

`basket` The basket in question

`voucher` The voucher in question

7.2.5 pre_payment

Raised immediately before attempting to take payment in the checkout.

Arguments sent with this signal:

`view` The view class instance

7.2.6 post_payment

Raised immediately after payment has been taken.

Arguments sent with this signal:

`view` The view class instance

7.2.7 order_placed

Raised by the `oscar.apps.order.utils.OrderCreator` class when creating an order.

Arguments sent with this signal:

`order` The order created

`user` The user creating the order (not necessarily the user linked to the order instance!)

CHAPTER 8

Contributing

Some ground rules:

- To avoid disappointment, new features should be discussed on the mailing list (django-oscar@googlegroups.com) before serious work starts.
- Pull requests will be rejected if sufficient tests aren't provided.
- Please update the documentation when altering behaviour or introducing new features.
- Follow the conventions (see below).

8.1 Installation

From zero to tests passing in 2 minutes (most of which is PIL installing):

```
git clone git@github.com:<username>/django-oscar.git
cd django-oscar
mkvirtualenv oscar
./setup.py develop
pip install -r requirements.txt
./run_tests.py
```

8.2 Writing docs

There's a helper script for building the docs locally:

```
cd docs
./test_docs.sh
```

8.3 Conventions

8.3.1 General

- PEP8 everywhere while remaining sensible

8.3.2 URLs

- List pages should use plurals, eg `/products/`, `/notifications/`
- Detail pages should simply be a PK/slug on top of the list page, eg `/products/the-bible/`, `/notifications/1/`
- Create pages should have 'create' as the final path segment, eg `/dashboard/notifications/create/`
- Update pages are sometimes the same as detail pages (ie when in the dashboard). In those cases, just use the detail convention, eg `/dashboard/notifications/3/`. If there is a distinction between the detail page and the update page, use `/dashboard/notifications/3/update/`.
- Delete pages, eg /dashboard/notifications/3/delete/

8.3.3 View class names

Classes should be named according to:

```
'%s%sView' % (class_name, verb)
```

For example, `ProductUpdateView`, `OfferCreateView` and `PromotionDeleteView`. This doesn't fit all situations but it's a good basis.

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*