django-oscar Documentation

Release 0.5

David Winterbottom

April 01, 2014

Contents

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 (e.g., 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.
- A range of merchandising blocks for promoting products throughout your site.
- Sophisticated offers that support virtually any kind of offer you can think of - multi-buys, bundles, buy X get 50% off Y etc
- Vouchers (built on top of the offers framework)
- Comprehensive dashboard
- Support for complex order processing such split payment orders, multi-batch shipping, order status pipelines.
- Extension libraries available for many payment gateways, including 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 (e.g., using a bankcard, voucher, gift card 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 subordinates.
- Multi-lingual products and categories.
- Digital products.
- Dynamically priced products (eg where the price is provided by an external service).

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.

First steps

1.1 Playing in the sandbox

Oscar ships with a 'sandbox' site, which is a vanilla install of Oscar using the default templates and styles. It is useful for exploring Oscar's functionality and developing new features.

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.

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

- All tax is set to zero
- No shipping methods are specified. The default is free shipping.
- No payment is required to submit an order as part of the checkout process

The sandbox is, in effect, the blank canvas upon which you can build your site.

1.1.1 Browse the external sandbox site

An instance of the sandbox site is build hourly from master branch and made available at http://latest.oscarcommerce.com

Warning: It is possible for users to access the dashboard and edit the site content. Hence, the data can get quite messy. It is periodically cleaned up.

Warning: Since this site is built from the unstable branch, occasionally things won't be fully stable. A more stable 'demo' site is in preparation, which will be more suitable for impressing clients/management.

1.1.2 Running the sandbox locally

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

Install Oscar and its dependencies within a virtualenv:

```
$ git clone git@github.com:tangentlabs/django-oscar.git
$ cd django-oscar
$ mkvirtualenv oscar
$ make sandbox
$ sites/sandbox/manage.py runserver
```

The sandbox site (initialised with a sample set of products) will be available at: http://localhost:8000. A sample superuser is installed with credentials:

```
username: superuser
email: superuser@example.com
password: testing
```

1.2 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.

1.2.1 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.

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`. It is also recommended to use `django.middleware.transaction.TransactionMiddleware` too

Now 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.apps.customer.notifications.context_processors.notifications',
    'oscar.core.context_processors.metadata',
)
```

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

```
from oscar import get_core_apps

INSTALLED_APPS = [
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.flatpages',
    ...
    'south',
    'compressor',
] + get_core_apps()
```

Note that Oscar requires django.contrib.messages and django.contrib.flatpages which aren't included by default.

Next, add django.contrib.flatpages.middleware.FlatpageFallbackMiddleware to your MIDDLEWARE_CLASSES setting:

```
MIDDLEWARE_CLASSES = (
    ...
    'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',
)
```

More info about django-flatpages installation at the django-project website.

Tip: Oscar's default templates use django-compressor but it's optional really. You may decide to use your own templates that don't use compressor. Hence why it is not one of the 'core apps'.

Now 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_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.simple_backend.SimpleEngine',
    },
}
```

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 *
```

URLs

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

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

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

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.

Defining the order pipeline

The order management in Oscar relies on the order pipeline that defines all the statuses an order can have and the possible transitions for any given status. Statuses in Oscar are not just used for an order but are handled on the line level as well to be able to handle partial shipping of an order.

The order status pipeline is different for every shop which means that changing it is fairly straightforward in Oscar. The pipeline is defined in your `settings.py` file using the `OSCAR_ORDER_STATUS_PIPELINE` setting. You also need to specify the initial status for an order and a line item in `OSCAR_INITIAL_ORDER_STATUS` and `OSCAR_INITIAL_LINE_STATUS` respectively.

To give you an idea of what an order pipeline might look like take a look at the Oscar sandbox settings:

```
OSCAR_INITIAL_ORDER_STATUS = 'Pending'
OSCAR_INITIAL_LINE_STATUS = 'Pending'
OSCAR_ORDER_STATUS_PIPELINE = {
    'Pending': ('Being processed', 'Cancelled',),
    'Being processed': ('Processed', 'Cancelled',),
    'Cancelled': (),
}
```

Defining the order status pipeline is simply a dictionary of where each status is given as a key. Possible transitions into other statuses can be specified as an iterable of status names. An empty iterable defines an end point in the pipeline.

With these three settings defined in your project you'll be able to see the different statuses in the order management dashboard.

1.2.2 Next steps

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

1.3 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 modeled 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.

1.3.1 Catalogue

What are your product types?

Are you selling books, DVDs, clothing, downloads, or fruit and vegetables? 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*

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 (e.g., 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? (e.g., 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*

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*

1.3.2 Pricing, stock and availability

How is tax calculated?

What availability messages are shown to customers?

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

- *How to configure stock messaging*

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.

1.3.3 Shipping

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*

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 (e.g., overseas orders have higher charges)
- The contents of the basket (e.g., free shipping for downloadable products)
- Who the user is (e.g., 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 determining which shipping methods are available to the user.

Recipes:

- *How to configure shipping*

1.3.4 Payment

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 gift cards.

Possible payment sources include:

- Bankcard
- Google checkout
- PayPal
- Business account
- Managed budget
- Gift card
- 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*

When will payment be taken?

A common pattern is to 'pre-auth' a bankcard at the point of checkout then 'settle' for the appropriate amounts 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?

1.4 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.

If you think you found a bug, please read *Reporting bugs* and report it in the GitHub issue tracker.

CHAPTER 2

Using Oscar

All you need to start developing an Oscar project.

2.1 Recipes

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

2.1.1 Customisation

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. To do it, you will need to let Oscar know that you're replacing the corresponding core app with yours. You can do that by supplying an extra argument to `get_core_apps` function:

```
# myproject/settings.py

from oscar import get_core_apps
# ...
INSTALLED_APPS = [
] + get_core_apps(['myproject.promotions'])
```

The `get_core_apps` function will replace `oscar.apps.promotions` with `myproject.promotions`.

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.

Now create the alternative template `new-homeview.html`. This could either be in a project-level `templates` folder that is added to your `TEMPLATE_DIRS` settings, or a app-level `templates` folder within your 'promotions' app. For now, put something simple in there, such as:

```
<html>
    <body>
        <p>You have successfully overridden the homepage template.</p>
    </body>
</html>
```

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.app 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

application = BaseApplication()
```

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

Django admin One pain point with replacing one of Oscar's apps with a local one in `INSTALLED_APPS` is that admin integration is lost from the original app. If you'd like to use the Django admin functionality you just need to run the register code in the replaced app's `admin.py`:

```
# myprojects/promotions/admin.py
import oscar.apps.promotions.admin
```

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

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 django.db import models

from oscar.apps.catalogue.abstract_models import AbstractProduct

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

from oscar.apps.catalogue.models import *
```

Make sure to import the remaining Oscar models at the bottom of your file.

The last thing you need to do now is make Django update the database schema and create a new column in the product table. We recommend to use South migrations for this (internally Oscar already uses it) so all you need to do is create a new schema migration. Depending on your setup you should follow one of these two options:

1. You **have not** run `./manage.py migrate` before

   You can simply generate a new initial migration using:

   ```
   ./manage.py schemamigration catalogue --initial
   ```

2. You **have** run `./manage.py migrate` before

   You have to copy the `migrations` directory from `oscar/apps/catalogue` (the same as the `models.py` you just copied) and put it into your `catalogue` app. Now create a new (additional) schemamigration using the `schemamigration` management command and follow the instructions:

   ```
   ./manage.py schemamigration catalogue --auto
   ```

To apply the migration you just created, all you have to do is run `./manage.py migrate catalogue` and the new column is added to the product table in the database.

Customising Products

You should inherit from `AbstractProduct` as above to alter behaviour for all your products. Further subclassing is not recommended, because using methods and attributes of concrete subclasses of `Product` are not available unless explicitly casted to that class. To model different classes of products, use `ProductClass` and `ProductAttribute` instead.

Model customisations are not picked up

It's a common problem that you're trying to customise one of Oscar's models, but your new fields don't seem to get picked up. That is usually caused by Oscar's models being imported before your customised ones. Django's model registration disregards all further model declarations.

In your overriding `models.py`, ensure that you import Oscar's models *after* your custom ones have been defined. If that doesn't help, you have an import from `oscar.apps.*.models` somewhere that is being executed before your models are parsed. One trick for finding that import: put `assert False` in the relevant Oscar's models.py, and the stack trace will show you the importing module.

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

How to override a core class

Much of Oscar's functionality is implemented using classes, when a module function might seem a better choice. This is to allow functionality to be customised. Oscar uses a dynamic class loading mechanism that can be used to override Oscar's core classes and use custom versions.

Example

Suppose you want to alter the way order numbers 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 (i.e., `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
```

`INSTALLED_APPS` tweak

You will need to add your app that contains the overriding class to `INSTALLED_APPS`, as well as let Oscar know that you're replacing the corresponding core app with yours. You can do that by supplying an extra argument to `get_core_apps` function:

```
# settings.py

from oscar import get_core_apps
# ...
INSTALLED_APPS = [
    # all your apps in here as usual, EXCLUDING yourproject.order
] + get_core_apps(['yourproject.order'])
```

The `get_core_apps` function will replace `oscar.apps.order` with `yourproject.order`.

Testing

You can test whether your overriding worked by trying to get a class from your module:

```
>>> from oscar.core.loading import get_class
>>> get_class('order.utils', 'OrderNumberGenerator')
```

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

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_MAIN_TEMPLATE_DIR
TEMPLATE_DIRS = (
    location('templates'),
    OSCAR_MAIN_TEMPLATE_DIR,
)
```

The `OSCAR_MAIN_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 'oscar/base.html' %}

...
```

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

Overriding individual products partials

Apart from overriding `catalogue/partials/product.html` to change the looks for all products, you can also override it for individual products by placing templates in `catalogue/partials/product/upc-%s.html` or `catalogue/partials/product/class-%s.html`, where `%s` is the product's UPC or product class in lower case, respectively.

Example: Changing the analytics package

Support you want to use an alternative analytics package to Google analytics. We can achieve this by overriding templates where the analytics urchin is loaded and called.

The main template `base.html` has a 'tracking' block which includes a Google Analytics partial. We want to replace this with our own code. To do this, create a new `base.html` in your project that subclasses the original:

```
# yourproject/templates/base.html
{% extends oscar/base.html %}

{% block tracking %}
<script type="javascript">
    ... [custom analytics here] ...
</script>
{% endblock %}
```

Doing this will mean all templates that inherit from `base.html` will include your custom tracking.

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.

How to change an existing URL pattern

Oscar has many views and associated URLs. Often you want to customised these URLs for your domain. For instance, you might want to use American spellings rather than British (`catalog` instead of `catalogue`).

URLs in Oscar

Oscar’s views and URLs use a tree of ‘app’ instances, each of which subclass `oscar.core.application.Application` and provide `urls` property. Oscar has a root app instance in `oscar/app.py` which can be imported into your `urls.py`:

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

urlpatterns = patterns('',
   ... # Your other URLs
   (r'', include(application.urls)),
)
```

Customising

In order to customise Oscar’s URLs, you need to use a custom app instance in your root `urls.py` instead of Oscar’s default instance. Hence, to use `catalog` instead of `catalogue`, create a subclass of Oscar’s main `Application` class and override the `get_urls` method:

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

class MyShop(app.Shop):
    # Override get_urls method
    def get_urls(self):
        urlpatterns = patterns('',
            (r'^catalog/', include(self.catalogue_app.urls)),

            ... # all the remaining URLs, removed for simplicity
        )
        return urlpatterns

application = MyShop()
```

Now modify your root `urls.py` to use your new application instance:

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

urlpatterns = patterns('',
   ... # Your other URLs
   (r'', include(application.urls)),
)
```

All URLs containing `catalogue` previously are now displayed as `catalog`.

How to configure the dashboard navigation

Oscar comes with a pre-configured dashboard navigation that gives you access to its individual pages. If you have your own dashboard app that you would like to show up in the dashboard navigation or want to arrange it differently, that's very easy. All you have to do is override the `OSCAR_DASHBOARD_NAVIGATION` setting in you settings file.

Add your own dashboard menu item

Assuming that you just want to append a new menu item to the dashboard, all you have to do is open up your settings file and somewhere below the import of the Oscar default settings:

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

add your custom dashboard configuration. Let's assume you would like to add a new item "Store Manager" with a submenu item "Stores". The way you would do that is:

```
OSCAR_DASHBOARD_NAVIGATION += [
    {
        'label': _('Store manager'),
        'children': [
            {
                'label': _('Stores'),
                'url_name': 'your-reverse-url-lookup-name',
            },
         ]
    },
]
```

That's it. You should now have *Store manager* > *Stores* in you dashboard menu.

Add an icon to your dashboard menu

Although you have your menu in the dashboard now, it doesn't look as nice as the other menu items that have icons displayed next to them. So you probably want to add an icon to your heading.

Oscar uses Font Awesome for its icons which makes it very simple to an icon to your dashboard menu. All you need to do is find the right icon for your menu item. Check out the icon list to find one.

Now that you have decided for an icon to use, all you need to do add the icon class for the icon to your menu heading:

```
OSCAR_DASHBOARD_NAVIGATION += [
    {
        'label': _('Store manager'),
        'icon': 'icon-map-marker',
        'children': [
            {
                'label': _('Stores'),
                'url_name': 'your-reverse-url-lookup-name',
            },
         ]
    },
]
```

You are not resticted to use Font Awesome icons for you menu heading. Other web fonts will work as well as long as they support the same markup:

```
<i class="icon-map-marker"></i>
```

The class is of the `<i>` is defined by the *icon* setting in the configuration of your dashboard navigation above.

2.1.2 Catalogue

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 > Woman > Skirts',
)
for breadcrumbs in categories:
    create_from_breadcrumbs(breadcrumbs)
```

How to model your catalogue

Related recipes:

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

Importing a catalogue

...

2.1.3 Pricing, stock and availability

How to enforce stock rules

You can enforce 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")
```

How to configure stock messaging

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

class `oscar.apps.partner.abstract_models.`**`AbstractStockRecord`**`(`**args*, ***kwargs*`)`

A basic stock record.

This links a product to a partner, together with price and availability information. Most projects will need to subclass this object to add custom fields such as lead_time, report_code, min_quantity.

We deliberately don't store tax information to allow each project to subclass this model and put its own fields for convey tax.

Parameters

- **product_id** (*OneToOneField*) – Product
- **partner_id** (*ForeignKey*) – Partner
- **partner_sku** (*CharField*) – Partner sku
- **price_currency** (*CharField*) – Currency
- **price_excl_tax** (*DecimalField*) – Price (excl. tax)
- **price_retail** (*DecimalField*) – Price (retail)
- **cost_price** (*DecimalField*) – Cost price
- **num_in_stock** (*PositiveIntegerField*) – Number in stock
- **low_stock_threshold** (*PositiveIntegerField*) – Low stock threshold
- **num_allocated** (*IntegerField*) – Number allocated
- **date_created** (*DateTimeField*) – Date created
- **date_updated** (*DateTimeField*) – Date updated

`availability`

Return a product's availability as a string that can be displayed to the user. For example, "In stock", "Unavailable".

availability_code
Return an product's availability as a code for use in CSS to add icons to the overall availability mark-up. For example, "instock", "unavailable".

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 code 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.

class `oscar.apps.partner.wrappers.`**`DefaultWrapper`**
Default stockrecord wrapper

availability(*stockrecord*)
Return an availability message for the passed stockrecord.

Parameters **stockrecord** (*oscar.apps.partner.models.StockRecord*) – stockrecord instance

availability_code(*stockrecord*)
Return a code for the availability of this product.

This is normally used within CSS to add icons to stock messages

Parameters **stockrecord** (*oscar.apps.partner.models.StockRecord*) – stockrecord instance

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"
```

2.1.4 Payment

How to integrate payment

Oscar is designed to be very flexible around payment. It supports paying for an order with multiple payment sources and settling these sources at different times.

Models

The payment app provides several models to track payments:

- `SourceType` - This is the type of payment source used (eg PayPal, DataCash, BrainTree). As part of setting up a new Oscar site you would create a SourceType for each of the payment gateways you are using.
- `Source` - A source of payment for a single order. This tracks how an order was paid for. The source object distinguishes between allocations, debits and refunds to allow for two-phase payment model. When an order is paid for by multiple methods, you create multiple sources for the order.
- `Transaction` - A transaction against a source. These models provide better audit for all the individual transactions associated with an order.

Example

Consider a simple situation where all orders are paid for by PayPal using their 'SALE' mode where the money is settled immediately (one-phase payment model). The project would have a 'PayPal' SourceType and, for each order, create a new `Source` instance where the `amount_debitted` would be the order total. A `Transaction` model with `txn_type=Transaction.DEBIT` would normally also be created (although this is optional).

This situation is implemented within the sandbox site for the django-oscar-paypal extension. Please use that as a reference.

See also the sandbox for django-oscar-datacash which follows a similar pattern.

Integration into checkout

By default, Oscar's checkout does not provide any payment integration as it is domain-specific. However, the core checkout classes provide methods for communicating with payment gateways and creating the appropriate payment models.

Payment logic is normally implemented by using a customised version of `PaymentDetailsView`, where the `handle_payment` method is overridden. This method will be given the order number and order total plus any custom keyword arguments initially passed to `submit` (as `payment_kwargs`). If payment is successful, then nothing needs to be returned. However, Oscar defines a few common exceptions which can occur:

- `oscar.apps.payment.exceptions.RedirectRequired` For payment integrations that require redirecting the user to a 3rd-party site. This exception class has a `url` attribute that needs to be set.
- `oscar.apps.payment.exceptions.UnableToTakePayment` For _anticipated_ payment problems such as invalid bankcard number, not enough funds in account - that kind of thing.
- `oscar.apps.payment.exceptions.PaymentError` For _unanticipated_ payment errors such as the payment gateway not responding or being badly configured.

When payment has completed, there's a few things to do:

- Create the appropriate `oscar.apps.payment.models.Source` instance and pass it to `add_payment_source`. The instance is passed unsaved as it requires a valid order instance to foreign key to. Once the order is placed (and an order instance is created), the payment source instances will be saved.
- Record a 'payment event' so your application can track which lines have been paid for. The `add_payment_event` method assumes all lines are paid for by the passed event type, as this is the normal situation when placing an order. Note that payment events don't distinguish between different sources.

For example:

```python
from oscar.apps.checkout import views
from oscar.apps.payment import models


# Subclass the core Oscar view so we can customise
class PaymentDetailsView(views.PaymentDetailsView):

    def handle_payment(self, order_number, total_incl_tax, **kwargs):
        # Talk to payment gateway.  If unsuccessful/error, raise a
        # PaymentError exception which we allow to percolate up to be caught
        # and handled by the core PaymentDetailsView.
        reference = gateway.pre_auth(order_number, total_incl_tax, kwargs['bankcard'])

        # Payment successful! Record payment source
        source_type, __ = models.SourceType.objects.get_or_create(
            name="SomeGateway")
        source = models.Source(
            source_type=source_type,
            amount_allocated=total_incl_tax,
            reference=reference)
        self.add_payment_source(source)

        # Record payment event
        self.add_payment_event('pre-auth', total_incl_tax)
```

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:

```python
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.

2.1.5 Shipping

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 (e.g., staff get cheaper shipping rates)
- the basket (e.g., shipping is charged based on the weight of the basket)
- the shipping address (e.g., 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.

2.1.6 Offers

How to create a custom range

Oscar ships with a range model that represents a set of products from your catalogue. Using the dashbaord, this can be configured to be:

1. The whole catalogue
2. A subset of products selected by ID/SKU (CSV uploads can be used to do this)
3. A subset of product categories

Often though, a shop may need merchant-specific ranges such as:

- All products subject to reduced-rate VAT
- All books by a Welsh author
- DVDs that have an exclamation mark in the title

These are contrived but you get the picture.

Custom range interface

A custom range must:

- have a `name` attribute
- have a `contains_product` method that takes a product instance and return a boolean
- have a `num_products` method that returns the number of products in the range or `None` if such a query would be too expensive.

Example:

```
class ExclamatoryProducts(object)
    name = "Products including a '!'"

    def contains_product(self, product):
        return "!" in product.title

    def num_products(self):
        return Product.objects.filter(title__icontains="!").count()
```

Create range instance

To make this range available to be used in offers, do the following:

```
from oscar.apps.offer.custom import create_range

create_range(ExclamatoryProducts)
```

Now you should see this range in the dashboard for ranges and offers. Custom ranges are not editable in the dashboard but can be deleted.

Deploying custom ranges

To avoid manual steps in each of your test/stage/production environments, use South's data migrations to create ranges.

How to create a custom offer condition

Oscar ships with several condition models that can be used to build offers. However, occasionally a custom condition can be useful. Oscar lets you build a custom condition class and register it so that it is available for building offers.

Custom condition interface

Custom condition classes must be proxy models, subclassing Oscar's main Condition class.

At a minimum, a custom condition must:

- have a `name` attribute
- have an `is_satisified` method that takes a basket instance and returns a boolean

It can also implement:

- a `can_apply_condition` method that takes a product instance and returns a boolean depending on whether the condition is applicable to the product.

- a `consume_items` method that marks basket items as consumed once the condition has been met.
- a `get_upsell_message` method that returns a message for the customer, letting them know what they would need to do to qualify for this offer.
- a `is_partially_satisfied` method that tests to see if the customer's basket partially satisfies the condition (ie when you might want to show them an upsel message)

Silly example:

```
from oscar.apps.offer import models

class BasketOwnerCalledBarry(models.Condition):
    name = "User must be called barry"

    class Meta:
        proxy = True

    def is_satisfied(self, basket):
        if not basket.owner:
            return False
        return basket.owner.first_name.lower() == 'barry'
```

Create condition instance

To make this condition available to be used in offers, do the following:

```
from oscar.apps.offer.custom import create_condition

create_range(BasketOwnerCalledBarry)
```

Now you should see this range in the dashboard when creating/updating an offer.

Deploying custom conditions

To avoid manual steps in each of your test/stage/production environments, use South's data migrations to create conditions.

How to create a custom benefit

Oscar ships with several offer benefits for building offers. There are three types:

- Basket discounts. These lead to a discount off the price of items in your basket.
- Shipping discounts.
- Post-order actions. These are benefits that don't affect your order total but trigger some action once the order is placed. For instance, if your site supports loyalty points, you might create an offer that gives 200 points when a certain product is bought.

Oscar also lets you create your own benefits for use in offers.

Custom benefits

A custom benefit can be used by creating a benefit class and registering it so it is available to be used.

A benefit class must by a proxy class and have the following methods:

```
from oscar.apps.offer import models


class MyCustomBenefit(models.Benefit):

    class Meta:
        proxy = True

    @property
    def description(self):
        """
        Describe what the benefit does.

        This is used in the dashboard when selecting benefits for offers.
        """

    def apply(self, basket, condition, offer=None):
        """
        Apply the benefit to the passed basket and mark the appropriate
        items as consumed.

        The condition and offer are passed as these are sometimes required
        to implement the correct consumption behaviour.

        Should return an instance of
        ``oscar.apps.offer.models.ApplicationResult``
        """

    def apply_deferred(self, basket):
        """
        Perform a 'post-order action' if one is defined for this benefit

        Should return a message indicating what has happend.  This will be
        stored with the order to provide audit of post-order benefits.
        """
```

As noted in the docstring, the `apply` method must return an instance of `oscar.apps.offer.models.ApplicationResult`. There are three subtypes provided:

- `oscar.apps.offer.models.BasketDiscount`. This takes an amount as it's constructor paramter.
- `oscar.apps.offer.models.ShippingDiscount`. This indicates that the benefit affects the shipping charge.
- `oscar.apps.offer.models.PostOrderAction`. This indicates that the benefit does nothing to the order total, but does fire an action once the order has been placed. It takes a single `description` paramter to its constructor which is a message that describes what action will be taken once the order is placed.

Here's an example of a post-order action benefit:

```
from oscar.apps.offer import models

class ChangesCustomersName(models.Benefit):

    class Meta:
        proxy = True

    description = "Changes customer's name"
    def apply(self, basket, condition, offer=None):
        # We need to mark all items from the matched condition as 'consumed'
        # so that they are unavailable to be used with other offers.
        condition.consume_items(basket, ())
        return models.PostOrderAction(
            "You will have your name changed to Barry!")

    def apply_deferred(self, basket):
        if basket.owner:
            basket.owner.first_name = "Barry"
            basket.owner.save()
            return "Your name has been changed to Barry!"
        return "We were unable to change your name as you are not signed in"
```

2.1.7 Appearance

How to change Oscar's appearance

This is a guide for Front-End Developers (FEDs) working on Oscar projects, not on Oscar itself. It is written with Tangent's FED team in mind but should be more generally useful for anyone trying to customise Oscar and looking for the right approach.

Overview

Oscar ships with a set of HTML templates and a collection of static files (eg images, javascript). Oscar's default CSS is generated from LESS files.

Templates Oscar's default templates use the mark-up conventions from Twitter's Bootstrap project.

LESS/CSS Oscar contains three main LESS files:

- *styles.less*
- *responsive.less*
- *dashboard.less*

These use the LESS files that ship with Twitter's Bootstrap project as well as some Oscar-specific styling.

A few other CSS files are used to provide styles for javascript libraries.

The core CSS files are not committed to source control to avoid duplication issues. However, they are included in the released Oscar packages.

By default, the CSS files are used rather than the Less ones. To use Less directly, set `USE_LESS = True` in your settings file. You will also need to ensure that the `lessc` executable is installed and is configured using a setting like:

```
COMPRESS_PRECOMPILERS = (
    ('text/less', 'lessc {infile} {outfile}'),
)
```

Using offline compression Django compressor also provides a way of running offline compression which can be used during deployment to automatically generate CSS files from your LESS files. To make sure that compressor is obeying the `USE_LESS` setting and is not trying to compress CSS files that are not available, the setting has to be passed into the `COMPRESS_OFFLINE_CONTEXT`. You should add something like this to your settings file:

```
COMPRESS_OFFLINE_CONTEXT = {
    # this is the only default value from compressor itself
    'STATIC_URL': 'STATIC_URL',
    'use_less': USE_LESS,
}
```

Javascript Oscar uses javascript for progressive enhancements.

For the customer-facing pages, Oscar uses:

- jQuery 1.7
- a few selected plugins to provide functionality for the content blocks that can be set-up.
- the Bootstrap javascript
- an Oscar-specific JS file (ui.js)

In the dashboard, Oscar uses all the JS assets from the customer side as well as:

- jQuery UI 1.8
- wysihtml5 for HTML textareas
- an Oscar specific JS file for dashboard functionality (dashboard.js)

Customistation

Customising templates Oscar ships with a complete set of templates (in `oscar/templates`). These will be available to an Oscar project but can be overridden or modified.

The templates use Twitter's Bootstrap conventions for class names and mark-up.

There is a separate recipe on how to do this.

Customising statics Oscar's static files are stored in `oscar/static`. When a Django site is deployed, the `collectstatic` command is run which collects static files from all installed apps and puts them in a single location (called the `STATIC_ROOT`). It is common for a separate HTTP server (like nginx) to be used to serve these files, setting its document root to `STATIC_ROOT`.

For an individual project, you may want to override Oscar's static files. The best way to do this is to have a statics folder within your project and to add it to the `STATICFILES_DIRS` setting. Then, any files which match the same path as files in Oscar will be served from your local statics folder instead. For instance, if you want to use a local version of `oscar/css/styles.css`, your could create a file:

```
yourproject/
    static/
        oscar/
            css/
                styles.css
```

and this would override Oscar's equivalent file.

To make things easier, Oscar ships with a management command for creating a copy of all of its static files. This breaks the link with Oscar's static files and means everything is within the control of the project. Run it as follows:

```
./manage.py oscar_fork_statics
```

This is the recommended approach for non-trivial projects.

Another option is simply to ignore all of Oscar's CSS and write your own from scratch. To do this, you simply need to adjust the layout templates to include your own CSS instead of Oscar's. For instance, you might override `base.html` and replace the 'less' block:

```
# project/base.html

{% block less %}
    <link rel="stylesheet" type="text/less" href="{{ STATIC_URL }}myproject/less/styles.less" />
{% endblock %}
```

2.1.8 Contributing

How do I translate Oscar?

Before doing any translation work, ensure you are famility with Django's i18n guidelines.

Contributing translations to Oscar

To submit a new set of translations for Oscar, do the following:

1. Fork the repo and install Oscar's repo then navigate to the `oscar` folder:

   ```
   git clone git@github.com:$USERNAME/django-oscar.git
   cd django-oscar/
   mkvirtualenv oscar
   make sandbox
   ```

2. Generate the message files for a given language:

   ```
   django-admin.py makemessages --locale=$LANGUAGE_CODE
   ```

3. Use the Rosetta functionality within the sandbox to add translations for the new messages files.:

   ```
   cd sites/sandbox
   ./manage.py runserver
   open http://localhost:8000/rosetta
   ```

4. Send a pull request!:

   ```
   git checkout -b new-translation
   git add oscar/locale
   git commit
   git push origin new-translation
   ```

Your work will be much appreciated.

Translating Oscar within your project

If Oscar does not provide translations for your language, or if you want to provide your own, do the following.

Within your project, create a locale folder and a symlink to Oscar so that `./manage.py makemessages` finds Oscar's translatable strings:

```
mkdir locale i18n
ln -s $PATH_TO_OSCAR i18n/oscar
./manage.py makemessages --symlinks --locale=de
```

This will create the message files that you can now translate.

2.2 Oscar Core Apps explained

Oscar is split up in multiple, mostly independent apps.

2.2.1 Address

The address app provides core address models - it doesn't provide any views or other functionality. Of the 5 abstract models, only 2 have a non-abstract version in `oscar.apps.address.models` - the others are used by the order app to provide shipping and billing address models.

Abstract models

class `oscar.apps.address.abstract_models.`**`AbstractAddress`**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

Superclass address object

This is subclassed and extended to provide models for user, shipping and billing addresses.

Parameters

- **title** (*CharField*) – Title
- **first_name** (*CharField*) – First name
- **last_name** (*CharField*) – Last name
- **line1** (*CharField*) – First line of address
- **line2** (*CharField*) – Second line of address
- **line3** (*CharField*) – Third line of address
- **line4** (*CharField*) – City
- **state** (*CharField*) – State/county
- **postcode** (*CharField*) – Post/zip-code
- **country_id** (*ForeignKey*) – Country
- **search_text** (*CharField*) – Search text - used only for searching addresses

`active_address_fields`()

Returns the non-empty components of the address, but merging the title, first_name and last_name into a single line.

`name`()

Return the full name

`populate_alternative_model`(*address_model*)

For populating an address model using the matching fields from this one.

This is used to convert a user address to a shipping address as part of the checkout process.

salutation()
Return the salutation

summary
Returns a single string summary of the address, separating fields using commas.

class oscar.apps.address.abstract_models.**AbstractShippingAddress**(**args*, ***kwargs*)
Bases: oscar.apps.address.abstract_models.AbstractAddress

A shipping address.

A shipping address should not be edited once the order has been placed - it should be read-only after that.

Parameters

- **title** (*CharField*) – Title
- **first_name** (*CharField*) – First name
- **last_name** (*CharField*) – Last name
- **line1** (*CharField*) – First line of address
- **line2** (*CharField*) – Second line of address
- **line3** (*CharField*) – Third line of address
- **line4** (*CharField*) – City
- **state** (*CharField*) – State/county
- **postcode** (*CharField*) – Post/zip-code
- **country_id** (*ForeignKey*) – Country
- **search_text** (*CharField*) – Search text - used only for searching addresses
- **phone_number** (*CharField*) – Phone number
- **notes** (*TextField*) – For example, leave the parcel in the wheelie bin if I'm not in.

order
Return the order linked to this shipping address

class oscar.apps.address.abstract_models.**AbstractUserAddress**(**args*, ***kwargs*)
Bases: oscar.apps.address.abstract_models.AbstractShippingAddress

A user's address. A user can have many of these and together they form an 'address book' of sorts for the user.

We use a separate model for shipping and billing (even though there will be some data duplication) because we don't want shipping/billing addresses changed or deleted once an order has been placed. By having a separate model, we allow users the ability to add/edit/delete from their address book without affecting orders already placed.

Parameters

- **title** (*CharField*) – Title
- **first_name** (*CharField*) – First name
- **last_name** (*CharField*) – Last name
- **line1** (*CharField*) – First line of address
- **line2** (*CharField*) – Second line of address
- **line3** (*CharField*) – Third line of address

- **line4** (*CharField*) – City
- **state** (*CharField*) – State/county
- **postcode** (*CharField*) – Post/zip-code
- **country_id** (*ForeignKey*) – Country
- **search_text** (*CharField*) – Search text - used only for searching addresses
- **phone_number** (*CharField*) – Phone number
- **notes** (*TextField*) – For example, leave the parcel in the wheelie bin if I'm not in.
- **user_id** (*ForeignKey*) – User
- **is_default_for_shipping** (*BooleanField*) – Default shipping address?
- **is_default_for_billing** (*BooleanField*) – Default billing address?
- **num_orders** (*PositiveIntegerField*) – Number of orders
- **hash** (*CharField*) – Address hash
- **date_created** (*DateTimeField*) – Date created

`generate_hash``()`
Returns a hash of the address summary

`is_default_for_billing` **= None**
Whether this address should be the default for billing.

`save``(`**args*, ***kwargs*`)`
Save a hash of the address fields

class `oscar.apps.address.abstract_models.`**`AbstractBillingAddress`**`(`**args*, ***kwargs*`)`
Bases: `oscar.apps.address.abstract_models.AbstractAddress`

Parameters

- **title** (*CharField*) – Title
- **first_name** (*CharField*) – First name
- **last_name** (*CharField*) – Last name
- **line1** (*CharField*) – First line of address
- **line2** (*CharField*) – Second line of address
- **line3** (*CharField*) – Third line of address
- **line4** (*CharField*) – City
- **state** (*CharField*) – State/county
- **postcode** (*CharField*) – Post/zip-code
- **country_id** (*ForeignKey*) – Country
- **search_text** (*CharField*) – Search text - used only for searching addresses

`order`
Return the order linked to this shipping address

class `oscar.apps.address.abstract_models.`**`AbstractCountry`**`(`**args*, ***kwargs*`)`
Bases: `django.db.models.base.Model`

International Organization for Standardization (ISO) 3166-1 Country list.

Parameters

- **iso_3166_1_a2** (*CharField*) – Iso 3166-1 alpha-2
- **iso_3166_1_a3** (*CharField*) – Iso 3166-1 alpha-3
- **iso_3166_1_numeric** (*PositiveSmallIntegerField*) – Iso 3166-1 numeric
- **name** (*CharField*) – Official name (caps)
- **printable_name** (*CharField*) – Country name
- **is_highlighted** (*BooleanField*) – Is highlighted
- **is_shipping_country** (*BooleanField*) – Is shipping country

2.2.2 Analytics

The `oscar.analytics` module provides a few simple models for gathering analytics data on products and users. It listens for signals from other apps, and creates/updates simple models which aggregate this data.

Such data is useful for auto-merchandising, calculating product scores for search and for personalised marketing for customers.

2.2.3 Checkout

Flow

The checkout process comprises the following steps:

1. **Gateway** - Anonymous users are offered the choice of logging in, registering, or checking out anonymously. Signed in users will be automatically redirected to the next step.
2. **Shipping address** - Enter or choose a shipping address.
3. **Shipping method** - Choose a shipping method. If only one shipping method is available then it is automatically chosen and the user is redirected onto the next step.
4. **Payment method** - Choose the method of payment plus any allocations if payment is to be split across multiple sources. If only one method is available, then the user is redirected onto the next step.
5. **Preview** - The prospective order can be previewed.
6. **Payment details** - If any sensitive payment details are required (e.g., bankcard number), then a form is presented within this step. This has to be the last step before submission so that sensitive details don't have to be stored in the session.
7. **Submission** - The order is placed.
8. **Thank you** - A summary of the order with any relevant tracking information.

Customisation

The checkout can be customised in many ways as you can extend and override and class from Oscar's core using the overriding core classes technique detailed previously.

2.2.4 Offers

Oscar ships with a powerful and flexible offers engine. It is based around the concept of 'conditional offers' - that is, a basket must satisfy some condition in order to qualify for a benefit.

Oscar's dashboard can be used to administer offers.

Structure

A conditional offer is composed of several components:

- Customer-facing information - this is the name and description of an offer. These will be visible on offer-browsing pages as well as within the basket and checkout pages.
- Availability - this determines when an offer is available.
- Condition - this determines when a customer qualifies for the offer (eg spend £20 on DVDs). There are various condition types available.
- Benefit - this determines the discount a customer receives. The discount can be against the basket cost or the shipping for an order.

Availability

An offer's availability can be controlled by several settings which can be used in isolation or combination:

- Date range - a date can be set, outside of which the offer is unavailable.
- Max global applications - the number of times and offer can be used can be capped. Note that an offer can be used multiple times within the same order so this isn't the same as limiting the number of orders that can use an offer.
- Max user applications - the number of times a particular user can use an offer. This makes most sense to use in sites that don't allow anonymous checkout as it could be circumvented by submitting multiple anonymous orders.
- Max basket applications - the number of times an offer can be used for a single basket/order.
- Max discount - the maximum amount of discount an offer can give across all orders. For instance, you might have a marketing budget of £10000 and so you could set the max discount to this value to ensure that once £10000 worth of benefit had been awarded, the offer would no longer be available. Note that the total discount would exceed £10000 as it would have to cross this threshold to disable the offer.

Conditions

There are 3 built-in condition types that can be created via the dashboard. Each needs to be linked with a range object, which is subset of the product catalogue. Ranges are created independently in the dashboard.

- Count-based - ie a customer must buy X products from the condition range
- Coverge-based - ie a customer must buy X DISTINCT products from the condition range. This can be used to create "bundle" offers.
- Value-based - ie a customer must spend X on products from the condition range

It is also possible to create custom conditions in Python and register these so they are available to be selected within the dashboard. For instance, you could create a condition that specifies that the user must have been registered for over a year to qualify for the offer.

Under the hood, conditions are defined by 3 attributes: a range, a type and a value.

Benefits

There are several types of built-in benefit, which fall into one of two categories: benefits that give a basket discount, and those that give a shipping discount.

Basket benefits:

- Fixed discount - ie get £5 off DVDs
- Percentage discount - ie get 25% off books
- Fixed price - ie get any DVD for £8
- Multibuy - ie get the cheapest product that meets the condition for free

Shipping benefits (these largely mirror the basket benefits):

- Fixed discount - ie £5 off shipping
- Percentage discount - ie get 25% off shipping
- Fixed price - ie get shipping for £8

Like conditions, it is possible to create a custom benefit. An example might be to allow customers to earn extra credits/points when they qualify for some offer. For example, spend £100 on perfume, get 500 credits (note credits don't exist in core Oscar but can be implemented using the 'accounts' plugin).

Under the hood, benefits are modelled by 4 attributes: a range, a type, a value and a setting for the maxinum number of basket items that can be affected by a benefit. This last settings is useful for limiting the scope of an offer. For instance, you can create a benefit that gives 40% off ONE products from a given range by setting the max affected items to 1. Without this setting, the benefit would give 40% off ALL products from the range.

Benefits are slightly tricky in that some types don't require a range and ignore the value of the max items setting.

Examples

Here's some example offers:

3 for 2 on books

1. Create a range for all books.
2. Use a **count-based** condition that links to this range with a value of 3.
3. Use a **multibuy** benefit with no value or range (multibuy benefits use the range of the condition and so don't need their own)

Spend £20 on DVDs, get 25% off

1. Create a range for all DVDs.
2. Use a **value-based** condition that links to this range with a value of 20.
3. Use a **percentage discount** benefit that links to this range and has a value of 25.

Buy 2 Lonely Planet books, get £5 off a Lonely Planet DVD

1. Create a range for Lonely Planet books and another for Lonely Planet DVDs
2. Use a **count-based** condition linking to the book range with a value of 2
3. Use a **fixed discount** benefit that links to the DVD range and has a value of 5.

More to come...

2.2.5 Promotions

Promotions are small blocks of content that can link through to other parts of this site. Examples include:

- A banner image shown on at the top of the homepage that links through to a new offer page
- A "pod" image shown in the right-hand sidebar of a page, linking through to newly merchandised page.
- A biography of an author (featuring an image and a block of HTML) shown at the top of the search results page when the search query includes the author's surname.

These are modeled using a base `promotion` model, which contains image fields, the link destination, and two "linking" models which link promotions to either a page URL or a particular keyword.

2.2.6 Shipping

Shipping can be very complicated. Depending on the domain, a wide variety of shipping scenarios are found in the wild. For instance, calculation of shipping costs can depend on:

- Shipping method (e.g., standard, courier)
- Shipping address
- Time of day of order (e.g., if requesting next-day delivery)
- Weight of items in basket
- Customer type (e.g., business accounts get discounted shipping rates)
- Offers and vouchers that give free or discounted shipping

Further complications can arise such as:

- Only making certain shipping methods available to certain customers
- Tax is only applicable in certain situations

Oscar can handle all of these shipping scenarios.

Shipping in Oscar

Shipping is handled using "method" objects which represent a means of shipping an order (e.g., "standard" or "next-day" delivery). Each method is essentially a named calculator that takes a basket and is able to calculate the shipping costs with and without tax.

For example, you may model "standard" delivery by having a calculator object that charges a fixed price for each item in the basket. The method object could be configured by passing the fixed price to be used for calculation.

Shipping within checkout

Shipping is first encountered by customers within the checkout flow, on the "shipping method" view.

It is the responsibility of this class to either:

1. Offer an a set of delivery methods for the customer to choose from, displaying the cost of each.
2. If there is only one method available, to construct the appropriate shipping method and set it within the checkout session context.

The `ShippingMethodView` class handles this behaviour. Its core implementation looks up a list of available shipping methods using the `oscar.shipping.repository.Repository` class. If there is only one, then this is written out to the session and a redirect is issued to the next step of the checkout. If more than one, then each available method is displayed so the customer can choose.

Default behaviour

Oscar ships with a simple model for calculating shipping based on a charge per order, and a charge per item. This is the `OrderAndItemLevelChargeMethod` class and is configured by setting the two charges used for the calculation. You can use this model to provide multiple methods - each identified by a code.

The core `Repository` class will load all defined `OrderAndItemLevelChargeMethod` models and make them available to the customer. If none are set, then a *FreeShipping* method object will be returned.

Shipping method classes

Each method object must subclass `ShippingMethod` from `oscar.shipping.methods` which provides the required interface. Note that the interface does not depend on the many other factors that can affect shipping (e.g., shipping address). The way to handle this is within your "factory" method which returns available shipping methods.

Writing your own shipping method

Simple really - follow these steps:

1. Subclass `oscar.shipping.methods.ShippingMethod` and implement the methods `basket_charge_incl_tax` and `basket_charge_excl_tax` for calculating shipping costs.
2. Override the default `shipping.repository.Repository` class and implement your domain logic for determining which shipping methods are returned based on the user, basket and shipping address passed in.

2.3 Oscar settings

This is a comprehensive list of all the settings Oscar provides. All settings are optional.

2.3.1 Display settings

OSCAR_SHOP_NAME

Default: `Oscar`

The name of your e-commerce shop site.

OSCAR_SHOP_TAGLINE

Default: `Domain-driven e-Commerce for Django`

The tagline that is displayed next to the shop name and in the browser title.

OSCAR_PARTNER_WRAPPERS

Default: {}

This is an important setting which defines availability for each fulfillment partner. The setting should be a dict from parter name to a path to a "wrapper" class. For example:

```
OSCAR_PARTNER_WRAPPERS = {
    'Acme': 'myproject.partners.AcmeWrapper'
    'Omnicorp': 'myproject.partners.OmnicorpWrapper'
}
```

The wrapper class should subclass `oscar.apps.partner.wrappers.DefaultWrapper` and override the appropriate methods to control availability behaviour.

OSCAR_RECENTLY_VIEWED_PRODUCTS

Default: 20

The number of recently viewed products to store.

OSCAR_PRODUCTS_PER_PAGE

Default: 20

The number of products to paginate by.

OSCAR_SEARCH_SUGGEST_LIMIT

Default: 10

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

OSCAR_PROMOTION_POSITIONS

Default:

```
OSCAR_PROMOTION_POSITIONS = (('page', 'Page'),
                             ('right', 'Right-hand sidebar'),
                             ('left', 'Left-hand sidebar'))
```

The choice of display locations available when editing a promotion. Only useful when using a new set of templates.

OSCAR_PROMOTION_MERCHANDISING_BLOCK_TYPES

Default:

```
COUNTDOWN, LIST, SINGLE_PRODUCT, TABBED_BLOCK = (
    'Countdown', 'List', 'SingleProduct', 'TabbedBlock')
OSCAR_PROMOTION_MERCHANDISING_BLOCK_TYPES = (
    (COUNTDOWN, "Vertical list"),
    (LIST, "Horizontal list"),
    (TABBED_BLOCK, "Tabbed block"),
    (SINGLE_PRODUCT, "Single product"),
)
```

Defines the available promotion block types that can be used in Oscar.

OSCAR_DASHBOARD_NAVIGATION

Default: see `oscar.defaults` (too long to include here).

A list of dashboard navigation elements.

2.3.2 Order settings

OSCAR_INITIAL_ORDER_STATUS

The initial status used when a new order is submitted. This has to be a status that is defined in the `OSCAR_ORDER_STATUS_PIPELINE`.

OSCAR_INITIAL_LINE_STATUS

The status assigned to a line item when it is created as part of an new order. It has to be a status defined in `OSCAR_ORDER_STATUS_PIPELINE`.

OSCAR_ORDER_STATUS_PIPELINE

Default: `{}`

The pipeline defines the statuses that an order or line item can have and what transitions are allowed in any given status. The pipeline is defined as a dictionary where the keys are the available statuses. Allowed transitions are defined as iterable values for the corresponding status.

A sample pipeline (as used in the Oscar sandbox) might look like this:

```
OSCAR_INITIAL_ORDER_STATUS = 'Pending'
OSCAR_INITIAL_LINE_STATUS = 'Pending'
OSCAR_ORDER_STATUS_PIPELINE = {
    'Pending': ('Being processed', 'Cancelled',),
    'Being processed': ('Processed', 'Cancelled',),
    'Cancelled': (),
}
```

OSCAR_ORDER_STATUS_CASCADE

This defines a mapping of status changes for order lines which 'cascade' down from an order status change.

For example:

```
OSCAR_ORDER_STATUS_CASCADE = {
    'Being processed': 'In progress'
}
```

With this mapping, when an order has it's status set to 'Being processed', all lines within it have their status set to 'In progress'. In a sense, the status change cascades down to the related objects.

Note that this cascade ignores restrictions from the `OSCAR_LINE_STATUS_PIPELINE`.

`OSCAR_LINE_STATUS_PIPELINE`

Default: `{}`

Same as `OSCAR_ORDER_STATUS_PIPELINE` but for lines.

2.3.3 Checkout settings

`OSCAR_ALLOW_ANON_CHECKOUT`

Default: `False`

Specifies if an anonymous user can buy products without creating an account first. If set to `False` users are required to authenticate before they can checkout (using Oscar's default checkout views).

`OSCAR_REQUIRED_ADDRESS_FIELDS`

Default: `('first_name', 'last_name', 'line1', 'city', 'postcode', 'country')`

List of form fields that a user has to fill out to validate an address field.

2.3.4 Review settings

`OSCAR_ALLOW_ANON_REVIEWS`

Default: `True`

This setting defines whether an anonymous user can create a review for a product without registering first. If it is set to `True` anonymous users can create product reviews.

`OSCAR_MODERATE_REVIEWS`

Default: `False`

This defines whether reviews have to be moderated before they are publicly available. If set to `False` a review created by a customer is immediately visible on the product page.

2.3.5 Communication settings

`OSCAR_EAGER_ALERTS`

Default: `True`

This enables sending alert notifications/emails instantly when products get back in stock by listening to stock record update signals this might impact performance for large numbers stock record updates. Alternatively, the management command `oscar_send_alerts` can be used to run periodically, e.g. as a cronjob. In this case instant alerts should be disabled.

OSCAR_SEND_REGISTRATION_EMAIL

Default: `True`

Sending out *welcome* messages to a user after they have registered on the site can be enabled or disabled using this setting. Setting it to `True` will send out emails on registration.

OSCAR_FROM_EMAIL

Default: `oscar@example.com`

The email address used as the sender for all communication events and emails handled by Oscar.

OSCAR_STATIC_BASE_URL

Default: `None`

A URL which is passed into the templates for communication events. It is not used in Oscar's default templates but could be used to include static assets (eg images) in a HTML email template.

2.3.6 Offer settings

OSCAR_OFFER_BLACKLIST_PRODUCT

Default: `None`

A function which takes a product as its sole parameter and returns a boolean indicating if the product is blacklisted from offers or not.

Example:

```
from decimal import Decimal as D

def is_expensive(product):
    if product.has_stockrecord:
        return product.stockrecord.price_incl_tax < D('1000.00')
    return False

# Don't allow expensive products to be in offers
OSCAR_OFFER_BLACKLIST_PRODUCT = is_expensive
```

OSCAR_OFFER_ROUNDING_FUNCTION

Default: Round down to the nearest hundredth of a unit using `decimal.Decimal.quantize`

A function responsible for rounding decimal amounts when offer discount calculations don't lead to legitimate currency values.

2.3.7 Basket settings

OSCAR_BASKET_COOKIE_LIFETIME

Default: 604800 (1 week in seconds)

The time to live for the basket cookie in seconds.

OSCAR_MAX_BASKET_QUANTITY_THRESHOLD

Default: `None`

The maximum number of products that can be added to a basket at once.

OSCAR_BASKET_COOKIE_OPEN

Default: `oscar_open_basket`

OSCAR_BASKET_COOKIE_SAVED

Default: `oscar_saved_basket`

2.3.8 Currency settings

OSCAR_DEFAULT_CURRENCY

Default: `GBP`

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

OSCAR_CURRENCY_LOCALE

Default: `None`

This can be used to customise currency formatting. The value will be passed to the `format_currency` function from the Babel library.

OSCAR_CURRENCY_FORMAT

Default: `None`

This can be used to customise currency formatting. The value will be passed to the `format_currency` function from the Babel library.

2.3.9 Upload/media settings

OSCAR_IMAGE_FOLDER

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

The location within the `MEDIA_ROOT` folder that is used to store product images. The folder name can contain date format strings as described in the Django Docs.

OSCAR_PROMOTION_FOLDER

Default: `images/promotions/`

The folder within `MEDIA_ROOT` used for uploaded promotion images.

OSCAR_MISSING_IMAGE_URL

Default: `image_not_found.jpg`

Copy this image from `oscar/static/img` to your `MEDIA_ROOT` folder. It needs to be there so Sorl can resize it.

OSCAR_UPLOAD_ROOT

Default: `/tmp`

The folder is used to temporarily hold uploaded files until they are processed. Such files should always be deleted afterwards.

2.3.10 Slug settings

OSCAR_SLUG_MAP

Default: `None`

A dictionary to map strings to more readable versions for including in URL slugs. This mapping is appled before the slugify function. This is useful when names contain characters which would normally be stripped. For instance:

```
OSCAR_SLUG_MAP = {
    'c++': 'cpp',
    'f#': 'fshared',
}
```

OSCAR_SLUG_FUNCTION

Default: `django.template.defaultfilters.slugify`

The slugify function to use. Note that is used within Oscar's slugify wrapper (in `oscar.core.utils`) which applies the custom map and blacklist.

Example:

```
def some_slugify(value)
    pass

OSCAR_SLUG_FUNCTION = some_slugify
```

OSCAR_SLUG_BLACKLIST

Default: `None`

A list of words to exclude from slugs.

Example:

```
OSCAR_SLUG_BLACKLIST = ['the', 'a', 'but']
```

2.3.11 Misc settings

OSCAR_COOKIES_DELETE_ON_LOGOUT

Default: `['oscar_recently_viewed_products',]`

Which cookies to delete automatically when the user logs out.

2.4 Signals

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

2.4.1 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

2.4.2 product_search

Raised when a search is performed.

Arguments sent with this signal:

`query` The search term

`user` The user in question

2.4.3 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

2.4.4 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

2.4.5 pre_payment

Raised immediately before attempting to take payment in the checkout.

Arguments sent with this signal:

`view` The view class instance

2.4.6 post_payment

Raised immediately after payment has been taken.

Arguments sent with this signal:

`view` The view class instance

2.4.7 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!)

2.4.8 post_checkout

Raised by the `oscar.apps.checkout.mixins.OrderPlacementMixin` class when a customer completes the checkout process.

`order` The order created

`user` The user who completed the checkout

`request` The request instance

`response` The response instance

2.4.9 review_created

Raised when a product detail page is viewed.

Arguments sent with this signal:

`review` The review that was created

`user` The user performing the action

`request` The request instance

`response` The response instance

2.5 Upgrading

This document explains some of the issues that can be encountered whilst upgrading Oscar.

Note: Detailed upgrade instructions for specific releases can be found on the Github wiki.

2.5.1 Migrations

Oscar uses South to provide migrations for its apps. But since Oscar allows an app to be overridden and its models extended, handling migrations can be tricky when upgrading.

Suppose a new version of Oscar changes the models of the 'shipping' app and includes the corresponding migrations. There are two scenarios to be aware of:

Migrating uncustomised apps

Apps that you aren't customising will upgrade trivially as your project will pick up the new migrations from Oscar directly.

For instance, if you have `oscar.apps.core.shipping` in your `INSTALLED_APPS` then you can simply run:

```
./manage.py migrate shipping
```

to migrate your shipping app.

Migrating customised apps

For apps that you are customising, you need to create a new migration that picks up the changes in the core Oscar models:

For instance, if you have an app `myproject.shipping` that replaces `oscar.apps.shipping` in your `INSTALLED_APPS` then you can simply run:

```
./manage.py schemamigration shipping --auto
```

to create the appropriate migration.

CHAPTER 3

The Oscar open-source project

Learn about the ideas behind Oscar and how you can contribute.

3.1 Oscar 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 achieved in several ways:

3.1.1 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 modeling 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.

3.1.2 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.

3.1.3 All views are class-based

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

3.1.4 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.

3.2 Release notes

3.2.1 Oscar 0.5 release notes

Welcome to Oscar 0.5!

These release notes cover the new features as well as upgrading advice.

Overview

The main aim of this release was to add functionality to offers but scope expanded over time to include many fixes and improvements. Whilst there aren't that many new features from a customer perspective, a great deal of work has gone into reworking Oscar's structure to be more extensible.

Thanks to all the contributors who helped with this release.

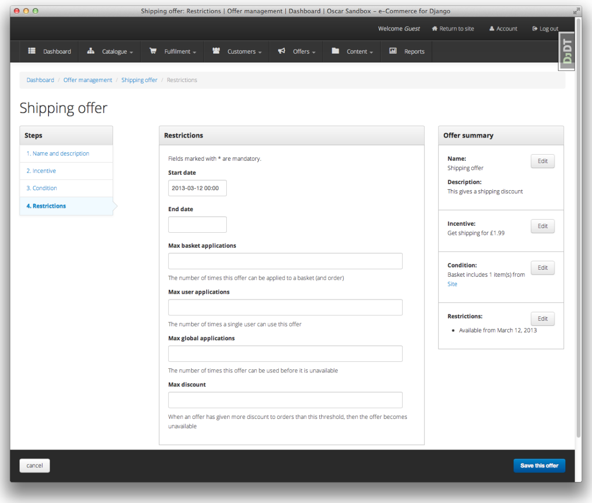
What's new in Oscar 0.5?

Offers++

Most of the new features in 0.5 are around offers.

- It is now possible to create custom ranges, conditions and benefits that can be used to create flexible offers. These ranges are created as Python classes conforming to a set interface which are registered at compile time to make them available in the dashboard.
- Offer benefits can now apply to the shipping charge for an order. Previously, all benefits were applied against the basket lines. There are three shipping benefits ready to use:

  - Fixed discount off shipping (eg get £5 off your shipping costs)
  - Percentage discount off shipping (eg get 25% off your shipping costs)
  - Fixed price shipping (eg your shipping charge will be £5)
- Offer benefits can now be deferred. That is, they don't affect either the basket lines nor the shipping charge. This is useful for creating benefits such as awarding loyalty points.
- Several new ways of restricting an offer's availability have been introduced:
  - An offer's lifetime can now be controlled to the second rather to the day (ie the relevant model fields are datetimes rather than dates). This makes it possibly to run offers for a small amount of time (eg for a single lunchtime).
  - An offer can be restricted to a max number of applications per *basket/order*. For example, an offer can configured so that it can only be used once in a single order.
  - An offer can be restricted to a max number of applications per *user*.
  - An offer can be restricted to a max number of *global* applications.
  - An offer can be restricted to give a maximum total discount. After this amount of discount has been awarded, the offer becomes unavailable.


Shipping offer: Restrictions | Offer management | Dashboard | Oscar Sandbox – e-Commerce for Django
Welcome *Guest*
Return to site
Account
Log out
Dashboard
Catalogue
Fulfilment
Customers
Offers
Content
Reports
DjDT
Dashboard / Offer management / Shipping offer / Restrictions
Edit
cancel
Save this offer

Shipping offer

Steps

1. Name and description
2. Incentive
3. Condition
4. **Restrictions**

Restrictions

Fields marked with * are mandatory.

Start date

2013-03-12 00:00

End date

Max basket applications

The number of times this offer can be applied to a basket (and order)

Max user applications

The number of times a single user can use this offer

Max global applications

The number of times this offer can be used before it is unavailable

Max discount

When an offer has given more discount to orders than this threshold, then the offer becomes unavailable

Offer summary

Name:
Shipping offer

Description:
This gives a shipping discount

Incentive:
Get shipping for £1.99

Condition:
Basket includes 1 item(s) from Site

Restrictions:
- Available from March 12, 2013

Figure 3.1: The restrictions editing page for an offer within the dashboard.

- Offers can now be suspended and reinstated.
- The offers dashboard has been rewritten.
- There is now an offers homepage that lists all active offers.

New dashboard skin

The design of the dashboard has been reworked, offering a better user experience throughout the dashboard. This work is still ongoing, further improvements in how the dashboard pages are laid out will appear in 0.6.

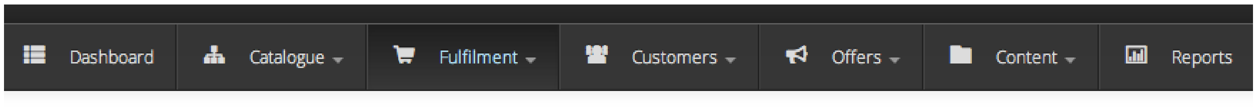
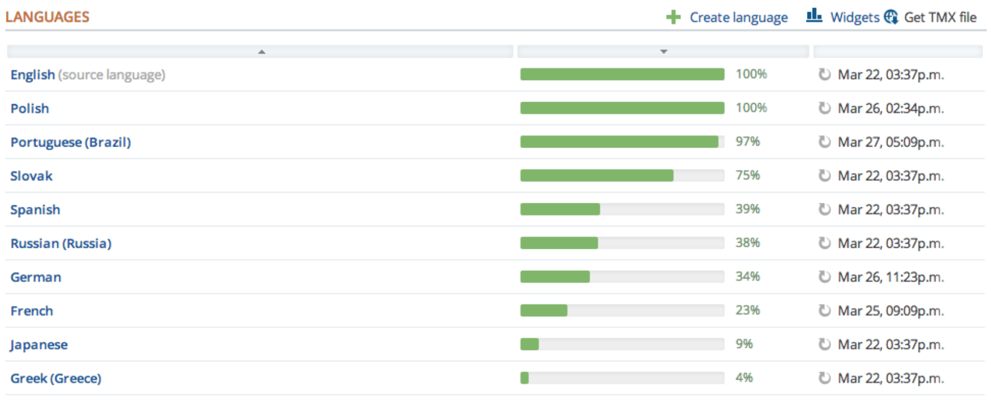


Figure 3.2: The new dashboard navigation.

Internationalisation

Oscar now uses Transifex to manage its translation files. Since 0.4, a considerable number of new languages are now supported (although many have partial coverage).


LANGUAGES   Create language   Widgets   Get TMX file

English (source language)	100%	Mar 22, 03:37p.m.
Polish	100%	Mar 26, 02:34p.m.
Portuguese (Brazil)	97%	Mar 27, 05:09p.m.
Slovak	75%	Mar 22, 03:37p.m.
Spanish	39%	Mar 22, 03:37p.m.
Russian (Russia)	38%	Mar 22, 03:37p.m.
German	34%	Mar 26, 11:23p.m.
French	23%	Mar 25, 09:09p.m.
Japanese	9%	Mar 22, 03:37p.m.
Greek (Greece)	4%	Mar 22, 03:37p.m.

Figure 3.3: A snippet from the Oscar Transifex page.

Oscar's default templates also now support a simple language picker.

New settings have been introduced to control how slugs are generated. By default, the unidecode package is used to gracefully handle non-ASCII chars in slugs.

Minor features

There are several noteworthy smaller improvements

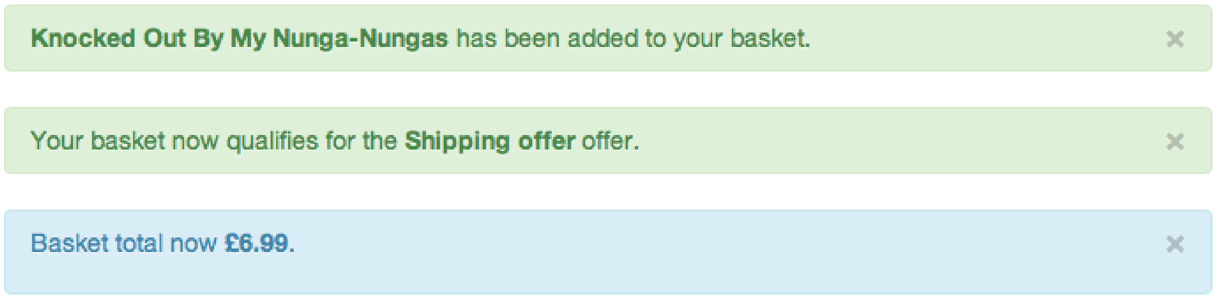
- The basket page now updates using AJAX rather than page reloads.
- Oscar's documentation has been reorganised and improved. This is part of an ongoing effort to improve it. Watch this space.
- Oscar's template now use django-compressor to compress CSS and JS assets.
- Products can now be deleted using the catalogue dashboard.
- Warnings emails are sent to customers when their password or email address is changed.
- Flash messages can now contain HTML.

Minor improvements

Several improvements have been made to ease development of Oscar (and Oscar projects):

- The sandbox can be configured to compile the LESS files directly. This is useful for developing Oscar's CSS/LESS files.

- A new management command `oscar_fork_statics` has been added to help with setting up static files for a new Oscar project.
- Alternative templates can now be used for different product classes in product browsing views.
- jQuery upgraded to 1.9.1
- Bootstrap upgraded to 2.3.1
- The test runner can now be run with tox.
- Oscar ships with profiling tools. There is a decorator and middleware available in `oscar.profiling` that can be used to help profile Oscar sites.
- Customers are notified if changes to their basket lead to new offers being applied (or if previously applied offers are no longer available).


Knocked Out By My Nunga-Nungas has been added to your basket. ×

Your basket now qualifies for the **Shipping offer** offer. ×

Basket total now **£6.99**. ×

Figure 3.4: A flash message indicating that the customer's basket has now qualified for a new offer.

- Some testing utilities have been extracted into a new package, django-oscar-testsupport, so they can be used by Oscar extensions.
- A Vagrant manifest is provided for testing Oscar against different database vendors.
- Oscar's javascript has been rewritten to be cleaner and more extensible.
- Coverage data is now submitted to coveralls.io

Upgrading

This section describes changes in core Oscar that you need to be aware of if you are upgrading from 0.4. See the *upgrading guidelines* for further details on the steps you need to take.

Migrations

There are new migrations in the following apps to be aware of.

- Address:
  - `0002`: Make `postcode` nullable on the `Address` model
- Catalogue:
  - `0009`: Add a `rating` field to the product model
  - `0010`: Populate the new `rating` field

Note: Note, if you are using a customised version of the catalogue app, then you should create a similar data migration to `0010` in your own project.

- Offer:
  - `0007`: Add `max_global_appliations` field to `ConditionalOffer` model
  - `0008`: Add `num_applications` field to `ConditionalOffer` model
  - `0009`: Rename `max_applications` field to `max_basket_applications`
  - `0010`: Add `max_user_applications` field to `ConditionalOffer` model
  - `0011`: Add `proxy_class` field to `Range` model
  - `0012`: Add `proxy_class` field to `Condition` model and make `range`, `type` and `value` nullable.
  - `0013`: Add unique index on `proxy_class` for the `Range` model
  - `0014`: Empty migration after branch merge
  - `0015`: Add `max_discount` field to `ConditionalOffer` model
  - `0016`: Add `status` field to `ConditionalOffer` model
  - `0017`: Change `start_date` and `end_date` to datetimes.
  - `0018`: Rename `start_date` and `end_date` to `start_datetime` and `end_datetime` respectively.
  - `0019`: Add `proxy_class` field to `Benefit` model and make `range`, `type` and `value` nullable.
- Order:
  - `0007`: Add `frequency` field to `OrderDiscount` model
  - `0008`: Add `category` field to `OrderDiscount` model
  - `0009`: Add `message` field to `OrderDiscount` model
- Partner:
  - `0004`: Add `code` field to `Partner` model
  - `0005`: Populate the new `code` field
  - `0006`: Add unique index on `code` field
  - `0007`: Remove unique index from `name` field and make nullable

Note: Note, if you are using a customised version of the partner app, then you should create a similar data migration to `0005` in your own project.

3.2.2 Oscar 0.5.1 release notes

This is a bugfix release for Oscar 0.5, backporting a few issues discovered during development of Oscar's demo site and fixing a couple of other bugs.

This release contains fixes for the following issues:

- The `is_available_to_buy` method was failing for variant products where the product class is defined on the parent product. Fixed in 7fd62f2af0 and … 80384a4007.

- The stockrecord partial `catalogue/partials/stock_record.html` incorrectly handled group products. Fixed in 5594bcccd6.
- The checkout thank-you template `checkout/thank_you.html` incorrectly looked up the line product URL. Fixed in cc5f63d827.
- The basket URL used for AJAX requests is no longer hard-coded. Fixed in . . . fd256b63b1.
- The dashboard voucher form now correctly validates when no start- or end-dates are entered. Fixed in 02b3644e3c
- The `AbstractStockRecord` model was not declared abstract. A migration has been added that cleans up the unnecessary database table. Fixed in . . . 610de82eb2

3.2.3 Oscar 0.5.2 release notes

This is Oscar 0.5.2, a security release for Oscar 0.5.

Insecure use of `SECRET_KEY` in basket cookie

For anonymous users, the basket ID is stored in a cookie. Previously, the value was signed using a simples CRC32 hash using the `SECRET_KEY`. However, a good rule of thumb is to never roll your own encryption, and it is possible that this method weakens the security of the `SECRET_KEY`.

The fix uses Django's cryptographic signing functionality to sign the cookie in a more secure manner.

3.2.4 Oscar 0.5.3 release notes

This is Oscar 0.5.3, a bug-fix release for Oscar 0.5.

The only change from 0.5.2 is to pin the dependency on Haystack to version 2.0.0. Previously, `setup.py` specified `2.0.0-beta` but this beta release has now been removed from PyPi, stopping Oscar from installing correctly.

3.3 Contributing to Oscar

You're thinking of helping out. That's brilliant - thank you for your time! You can contribute in many ways:

- Join the django-oscar mailing list and answer questions.
- *Report bugs* in our ticket tracker.
- *Submit pull requests* for new and/or fixed behavior.
- *Improve the documentation*.
- *Write tests*.
- Translations can be contributed using Transifex. Just apply for a language and go ahead!

3.3.1 Overview

Setting up the development environment

Fork the repo and run:

```
git clone git@github.com:<username>/django-oscar.git
cd django-oscar
mkvirtualenv oscar  # using virtualenvwrapper
make install
```

The `sandbox` site can be used to test our changes in a browser. It is easily created with `make sandbox`.

Writing LESS/CSS

Oscar's CSS files are build using LESS. However, the sandbox defaults to serving CSS files directly, bypassing LESS compilation.

If you want to develop the LESS files, set:

```
USE_LESS = True
COMPRESS_ENABLED = False
```

in `sites/sandbox/settings_local.py`. This will cause Oscar to use django-compressor to compile the LESS files as they are requested. For this to work, you will need to ensure that the LESS compiler `lessc` is installed. This can be acheived by running:

```
pip install -r requirements_less.txt
```

which will install the virtual-node and virtual-less packages, which will install node.js and LESS in your virtualenv.

If you have npm installed already, you install LESS using:

```
npm install less
```

Warning: If you do submit a pull request that changes the LESS files. Please also recompile the CSS files and include them in your pull request.

Vagrant

Oscar ships with a Vagrant virtual machine that can be used to test integration with various services in a controlled environment. For instance, it is used to test that the migrations run correctly in both MySQL and Postgres.

Building the Vagrant machine To create the machine, first ensure that Vagrant and puppet are installed. You will require a puppet version that supports `puppet module install`, that is > 2.7.14. Now run:

```
make puppet
```

to fetch the required puppet modules for provisioning. Finally, run:

```
vagrant up
```

to create the virtual machine and provision it.

Testing migrations against MySQL and Postgres To test the migrations against MySQL and Postgres, do the following:

1. SSH onto the VM:

   ```
   vagrant ssh
   ```

2. Change to sandbox folder and activate virtualenv:

   ```
   cd /vagrant/sites/sandbox
   source /var/www/virtualenv/bin/activate
   ```

3. Run helper script:

   ```
   ./test_migrations.sh

   This will recreate the Oscar database in both MySQL and Postgres and rebuild
   it using ``syncdb`` and ``migrate``.
   ```

Testing WSGI server configurations You can browse the Oscar sandbox site in two ways:

- Start Django's development server on port 8000:

  ```
  vagrant ssh
  cd /vagrant/sites/sandbox
  source /var/www/virtualenv/bin/activate
  ./manage.py runserver 0.0.0.0:8000
  ```

  The Vagrant machine forwards port 8000 to post 8080 and so the site can be accessed at http://localhost:8080 on your host machine.

- The Vagrant machine installs Apache2 and mod_wsgi. You can browse the site through Apache at http://localhost:8081 on your host machine.

Reporting bugs and requesting features

Before reporting a bug or requesting a new feature, please consider these general points:

- Check that someone hasn't already filed the bug or feature request by searching in the ticket tracker.
- Don't use the ticket system to ask support questions. Use the django-oscar mailing list for that.
- Don't use the ticket tracker for lengthy discussions, because they're likely to get lost. If a particular ticket is controversial, please move the discussion to django-oscar.

All bugs are reported on our GitHub issue tracker.

Reporting bugs

Well-written bug reports are *incredibly* helpful. However, there's a certain amount of overhead involved in working with any bug tracking system so your help in keeping our ticket tracker as useful as possible is appreciated. In particular:

- **Do** ask on django-oscar *first* if you're not sure if what you're seeing is a bug.
- **Do** write complete, reproducible, specific bug reports. You must include a clear, concise description of the problem, and a set of instructions for replicating it. Add as much debug information as you can: code snippets, test cases, exception backtraces, screenshots, etc. A nice small test case is the best way to report a bug, as it gives us an easy way to confirm the bug quickly.

Reporting user interface bugs and features

If your bug or feature request touches on anything visual in nature, there are a few additional guidelines to follow:

- Include screenshots in your ticket which are the visual equivalent of a minimal testcase. Show off the issue, not the crazy customizations you've made to your browser.
- If you're offering a pull request which changes the look or behavior of Oscar's UI, please attach before *and* after screenshots/screencasts.
- Screenshots don't absolve you of other good reporting practices. Make sure to include URLs, code snippets, and step-by-step instructions on how to reproduce the behavior visible in the screenshots.

Requesting features

We're always trying to make Oscar better, and your feature requests are a key part of that. Here are some tips on how to make a request most effectively:

- First request the feature on the django-oscar list, not in the ticket tracker. It'll get read more closely if it's on the mailing list. This is even more important for large-scale feature requests. We like to discuss any big changes to Oscar's core on the mailing list before actually working on them.
- Describe clearly and concisely what the missing feature is and how you'd like to see it implemented. Include example code (non-functional is OK) if possible.
- Explain *why* you'd like the feature, because sometimes it isn't obvious why the feature would be useful.

As with most open-source projects, code talks. If you are willing to write the code for the feature yourself or, even better, if you've already written it, it's much more likely to be accepted. Just fork Oscar on GitHub, create a feature branch, and show us your work!

Coding Style

General

Please follow these conventions while remaining sensible:

- PEP8 – Style Guide for Python Code
- PEP257 – Docstring Conventions
- Django Coding Style

Code Like a Pythonista is recommended reading.

URLs

- List pages should use plurals; e.g. `/products/`, `/notifications/`
- Detail pages should simply be a PK/slug on top of the list page; e.g. `/products/the-bible/`, `/notifications/1/`
- Create pages should have 'create' as the final path segment; e.g. `/dashboard/notifications/create/`
- Update pages are sometimes the same as detail pages (i.e., 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; e.g., `/dashboard/notifications/3/delete/`

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.

Referencing managers

Use `_default_manager` rather than `objects`. This allows projects to override the default manager to provide domain-specific behaviour.

Submitting pull requests

- To avoid disappointment, new features should be discussed on the mailing list (django-oscar@googlegroups.com) before serious work starts.
- Write tests! Pull requests will be rejected if sufficient tests aren't provided.
- Write docs! Please update the documentation when altering behaviour or introducing new features.
- Write good commit messages: see Tim Pope's excellent note.
- Make pull requests against Oscar's master branch unless instructed otherwise.

Test suite

Running tests

Oscar uses a nose testrunner which can be invoked using:

```
$ ./runtests.py
```

To run a subset of tests, you can use filesystem or module paths. These two commands will run the same set of tests:

```
$ ./runtests.py tests/unit/order
$ ./runtests.py tests.unit.order
```

To run an individual test class, use one of:

```
$ ./runtests.py tests/unit/order:TestSuccessfulOrderCreation
$ ./runtests.py tests.unit.order:TestSuccessfulOrderCreation
```

(Note the ':'.)

To run an individual test, use one of:

```
$ ./runtests.py tests/unit/order:TestSuccessfulOrderCreation.test_creates_order_and_line_models
$ ./runtests.py tests.unit.order:TestSuccessfulOrderCreation.test_creates_order_and_line_models
```

Testing against different setups

To run all tests against multiple versions of Django and Python, use tox:

```
$ tox
```

You need to have all Python interpreters to test against installed on your system. All other requirements are downloaded automatically.

Kinds of tests

Tests are split into 3 folders:

- unit - These are for tests that exercise a single unit of functionality, like a single model. Ideally, these should not write to the database at all - all operations should be in memory.
- integration - These are for tests that exercise a collection or chain of units, like testing a template tag.
- functional - These should be as close to "end-to-end" as possible. Most of these tests should use WebTest to simulate the behaviour of a user browsing the site.

Naming tests

Oscar's testrunner uses the progressive plugin when running all tests, but uses the spec plugin when running a subset. It is a good practice to name your test cases and methods so that the spec output reads well. For example:

```
$ ./runtests.py tests/unit/offer/benefit_tests.py:TestAbsoluteDiscount
nosetests --verbosity 1 tests/unit/offer/benefit_tests.py:TestAbsoluteDiscount -s -x --with-spec
Creating test database for alias 'default'...

Absolute discount
- consumes all lines for multi item basket cheaper than threshold
- consumes all products for heterogeneous basket
- consumes correct quantity for multi item basket more expensive than threshold
- correctly discounts line
- discount is applied to lines
- gives correct discount for multi item basket cheaper than threshold
- gives correct discount for multi item basket more expensive than threshold
- gives correct discount for multi item basket with max affected items set
- gives correct discount for single item basket cheaper than threshold
- gives correct discount for single item basket equal to threshold
- gives correct discount for single item basket more expensive than threshold
- gives correct discounts when applied multiple times
- gives correct discounts when applied multiple times with condition
- gives no discount for a non discountable product
- gives no discount for an empty basket

----------------------------------------------------------------------
Ran 15 tests in 0.295s
```

Writing documentation

The docs are built by calling `make docs` from your Oscar directory. They live in `/docs/source`. This directory structure is a simplified version of what Django does.

- `internals/` contains everything related to Oscar itself, e.g. contributing guidelines or design philosophies.

- `ref/` is the reference documentation, esp. consisting of
- `ref/apps/` which should be a guide to each Oscar core app, explaining it's function, the main models, how it relates to the other apps, etc.
- `topics/` will contain "meta" articles, explaining how things tie together over several apps, or how Oscar can be combined with other solutions.
- `howto/` contains tutorial-style descriptions on how to solve a certain problem.

`/index.rst` is designed as the entry point, and diverges from above structure to make the documentation more approachable. Other `index.rst` files should only be created if there's too many files to list them all. E.g. `/index.rst` directly links to all files in `topics/` and `internals/`, but there's an `index.rst` both for the files in `howto/` and `ref/apps/`.