django-oscar Documentation

Release 1.5

David Winterbottom

May 23, 2018

Contents


oscar

CHAPTER 1

Domain-driven e-commerce for Django

Oscar is an e-commerce framework for building domain-driven applications. It has flexibility baked into its core so that complicated requirements can be elegantly captured. You can tame a spaghetti domain without writing spaghetti code.

Years of e-commerce hard-earned experience informs Oscar's design.

Oscar is "domain-driven" in the sense that the core business objects can be customised to suit the domain at hand. In this way, your application can accurately capture the subtleties of its domain, making feature development and maintenance much easier.

Features:

- Any product type can be handled including downloadable products, subscriptions, child products (e.g., a T-shirt in different sizes and colours).
- Customisable products, such as T-shirts with personalised messages.
- Large catalogue support - 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 that replaces the Django admin completely
- 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 projects already handle:

- Paying for an order with multiple payment sources (e.g., using a bankcard, voucher, gift card and points 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 used in production in several applications to sell everything from beer mats to iPads. The source is on GitHub - contributions are always welcome.

First steps

Sample Oscar projects

Oscar ships with one sample project: a 'sandbox' site, which is a vanilla install of Oscar using the default templates and styles.

The sandbox site

The sandbox site is a minimal implementation of Oscar where everything is left in its default state. It is useful for exploring Oscar's functionality and developing new features.

It only has one notable customisation on top of Oscar's core:

- 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 which will be automatically selected during checkout (as it's the only option).
- 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.

Browse the external sandbox site

An instance of the sandbox site is built 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.

Run the sandbox locally

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

Warning: While installing Oscar is straightforward, some of Oscar's dependencies don't support Windows and are tricky to be properly installed, and therefore you might encounter some errors that prevent a successful installation.

Install Oscar and its dependencies within a virtualenv:

```
$ git clone https://github.com/django-oscar/django-oscar.git
$ cd django-oscar
$ mkvirtualenv oscar  # needs virtualenvwrapper
(oscar) $ make sandbox
(oscar) $ sandbox/manage.py runserver
```

Warning: Note, these instructions will install the head of Oscar's 'master' branch, not an official release. Occasionally the sandbox installation process breaks while support for a new version of Django is being added (often due dependency conflicts with 3rd party libraries). Please ask on the mailing list if you have problems.

If you do not have `mkvirtualenv`, then replace that line with:

```
$ virtualenv oscar
$ source ./oscar/bin/activate
(oscar) $
```

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
```

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.

Install Oscar and its dependencies

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

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

If you do not have mkvirtualenv, then replace that line with:

```
$ virtualenv oscar
$ . ./oscar/bin/activate
(oscar) $
```

This will create a folder `frobshop` for your project. It is highly recommended to install Oscar in a virtualenv.

Attention: Please ensure that `pillow`, a fork of the the Python Imaging Library (PIL), gets installed with JPEG support. Supported formats are printed when `pillow` is first installed. Instructions on how to get JPEG support are highly platform specific, but guides for `PIL` should work for `pillow` as well. Generally speaking, you need to ensure that `libjpeg-dev` is installed and found during installation.

Django settings

First, edit your settings file `frobshop.frobshop.settings.py` to import all of Oscar's default settings.

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

Now modify your `TEMPLATES` to include the main Oscar template directory and add the extra context processors.

```
from oscar import OSCAR_MAIN_TEMPLATE_DIR

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(BASE_DIR, 'templates'),
            OSCAR_MAIN_TEMPLATE_DIR
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.template.context_processors.i18n',
                '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',
            ],
        },
    },
]
```

Attention: Before Django 1.8 this setting was split between `TEMPLATE_CONTEXT_PROCESSORS` and `TEMPLATE_DIRS`.

Next, modify `INSTALLED_APPS` to be a list, add `django.contrib.sites`, `django.contrib.flatpages`, and `widget_tweaks` and append Oscar's core apps. Also set `SITE_ID`:

```
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.staticfiles',
    'django.contrib.flatpages',
    ...
    'compressor',
    'widget_tweaks',
] + get_core_apps()

SITE_ID = 1
```

Note that Oscar requires `django.contrib.flatpages` which isn't included by default. `flatpages` also requires `django.contrib.sites`. More info about installing `flatpages` is in the Django docs.

Tip: Oscar's default templates use django-compressor and django-widget-tweaks but it's optional really. You may decide to use your own templates that don't use either. Hence why they are not in the 'core apps'.

Next, add `oscar.apps.basket.middleware.BasketMiddleware` and `django.contrib.flatpages.middleware.FlatpageFallbackMiddleware` to your `MIDDLEWARE_CLASSES` setting.

```
MIDDLEWARE_CLASSES = (
    ...
    'oscar.apps.basket.middleware.BasketMiddleware',
    'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',
)
```

Note: In django 1.10 and above: make sure to replace `MIDDLEWARE` with `MIDDLEWARE_CLASSES`

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.

Ensure that your media and static files are configured correctly. This means at the least setting `MEDIA_URL` and `STATIC_URL`. If you're serving files locally, you'll also need to set `MEDIA_ROOT` and `STATIC_ROOT`. Check out the sandbox settings for a working example. If you're serving files from a remote storage (e.g. Amazon S3), you must manually copy a *"Image not found" image* into `MEDIA_ROOT`.

URLs

Alter your `frobshop/urls.py` to include Oscar's URLs. You can also include the Django admin for debugging purposes. But please note that Oscar makes no attempts at having that be a workable interface; admin integration exists to ease the life of developers.

If you have more than one language set your Django settings for `LANGUAGES`, you will also need to include Django's i18n URLs:

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

urlpatterns = [
    url(r'^i18n/', include('django.conf.urls.i18n')),

    # The Django admin is not officially supported; expect breakage.
    # Nonetheless, it's often useful for debugging.
    url(r'^admin/', include(admin.site.urls)),

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

Search backend

If you're happy with basic search for now, you can just add Haystack's simple backend to the `HAYSTACK_CONNECTIONS` option in your Django settings:

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

Oscar uses Haystack to abstract away from different search backends. Unfortunately, writing backend-agnostic code is nonetheless hard and Apache Solr is currently the only supported production-grade backend. Your Haystack config could look something like this:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://127.0.0.1:8983/solr',
        'INCLUDE_SPELLING': True,
    },
}
```

Oscar includes a sample schema to get started with Solr. More information can be found in the *recipe on getting Solr up and running*.

Database

Check your database settings. A quick way to get started is to use SQLite:

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

Note that we recommend using `ATOMIC_REQUESTS` to tie transactions to requests.

Create database

Oscar ships with migrations. Django's migration framework will detect them automatically and will do the right thing. Create the database and the shop should be browsable:

```
$ python manage.py migrate
$ python manage.py runserver
```

You should now have an empty, but running Oscar install that you can browse at http://localhost:8000.

Initial data

The default checkout process requires a shipping address with a country. Oscar uses a model for countries with flags that indicate which are valid shipping countries and so the `country` database table must be populated before a customer can check out.

The easiest way to achieve this is to use country data from the pycountry package. Oscar ships with a management command to parse that data:

```
$ pip install pycountry
[...]
$ python manage.py oscar_populate_countries
```

By default, this command will mark all countries as a shipping country. Call it with the `--no-shipping` option to prevent that. You then need to manually mark at least one country as a shipping country.

Creating product classes and fulfillment partners

Every Oscar deployment needs at least one *`product class`* and one *`fulfillment partner`*. These aren't created automatically as they're highly specific to the shop you want to build.

When managing your catalogue you should always use the Oscar dashboard, which provides the necessary functionality. Use your Django superuser email and password to login to: http://127.0.0.1:8000/dashboard/ and create instances of both there.

It is important to note that the Django admin site is not supported. It may or may not work and is only included in the sandbox for developer's convenience.

For a deployment setup, we recommend creating product classes as data migration.

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.

Next steps

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

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 foundations upon which 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.

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. Modelling the catalogue on the backend is explained in *Modelling your catalogue*

How is your catalogue organised?

How are products organised on the front end? 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 are products managed?

Is the catalogue managed by an 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 or feature*

Pricing, stock and availability

How is tax calculated?

Taxes vary widely between countries. Even the way that prices are displayed varies between countries. For instance, in the UK and Europe prices are shown inclusive of VAT whereas in the US, taxes are often not shown until the final stage of checkout.

Furthermore, the amount of tax charged can vary depending on a number of factors, including:

- The products being bought (eg in the UK, certain products have pay no VAT).
- Who the customer is. For instance, sales reps will often not pay tax whereas regular customers will.
- The shipping address of the order.
- The payment method used.

Recipes: * *How to handle US taxes*

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

A 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.

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*

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 suitably 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).

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, or legal restrictions of the country you are operating in.

- 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?

Order processing

How will orders be processed?

Orders can be processed in many ways, including:

- Manual process. For instance, a worker in a warehouse may download a picking slip from the dashboard and mark products as shipped when they have been put in the van.
- Fully automated process, where files are transferred between the merchant and the fulfillment partner to indicate shipping statuses.

Recipes:

- *How to set up order processing*

Modelling your catalogue

Oscar gives you several layers of modelling your products.

Note that this document is merely concerned with how to model products in your database. How you display them in your front-end, e.g. in a category tree, is out of scope.

Product classes

Typical examples for product classes would be: T-shirts, Books, Downloadable products.

Each product is assigned to exactly one product class.

Settings on a product class decide whether stock levels are *tracked* for the associated products, and whether they *require shipping*.

Furthermore, they govern what kind of product attributes can be stored on the products. We'll get to attributes in a bit, but think T-shirt sizes, colour, number of pages, etc.

Typically stores will have between 1 and maybe 5 product classes.

Product attributes

Product attributes let you set additional data on a product without having to customise the underlying Django models. There's different types of attributes, e.g. ones for just associating text (type `text` or `richtext`), for related images and files (type `image` and `file`), etc.

Storing data in structured attributes also makes it easy to search and filter products based on specific attributes.

The available product attributes for a product are set when creating the product's class. The sandbox comes with a product class for T-shirts, and they have a `size` attribute:

```
> shirt = Product.objects.filter(product_class__slug='t-shirt').first()
> shirt.attr.size
<AttributeOption: Large>
```

You can assign `option` s to your product. For example you want a Language attribute to your product, and a couple of options to choose from, for example English and Croatian. You'd first create an `AttributeOptionGroup` that would contain all the `AttributeOption` s you want to have available:

```
> language = AttributeOptionGroup.objects.create(name='Language')
```

Assign a couple of options to the Language options group:

```
> AttributeOption.objects.create(
>     group=language,
>     option='English'
> )
> AttributeOption.objects.create(
>     group=language,
>     option='Croatian'
> )
```

Finally assign the Language options group to your product as an attribute:

```
> klass = ProductClass.objects.create(name='foo', slug='bar')
> ProductAttribute.objects.create(
>     product_class=klass,
>     name='Language',
>     code='language',
>     type='option',
>     option_group=language
> )
```

You can go as far as associating arbitrary models with it. Use the `entity` type:

```
> klass = ProductClass.objects.create(name='foo', slug='bar')
> ProductAttribute.objects.create(
      product_class=klass, name='admin user', code='admin_user', type='entity')
<ProductAttribute: admin user>
> p = Product(product_class=klass)

> p.attr.admin_user = User.objects.first()
> p.save()
> p.attr.admin_user
<User: superuser>
```

You can also use the `multi_option` attribute type if your options are not mutually exclusive.

> klass = ProductClass.objects.create(name='foo', slug='bar') > ProductAttribute.objects.create(> product_class=klass, > name='Size', > code='size', > type='multi_option', > option_group=language >)

This will let you assign multiple values (`size` in the example above) to the attribute.

All attribute types apart from `entity` can be edited in the product dashboard. The latter is too dependent on your use case and you will need to decide yourself how you want to set and display it.

Parent and child products

Often there's an overarching product, which groups other products. In that case, you can create a parent product, and then set the `parent` field on the child products. By default, only parent products (or products without children) get their own URL. Child products inherit their product class from the parent, and only child products can have stock records (read: pricing information) on them.

Going further

Oscar's modelling options don't stop there. If the existing framework does not suit your need, you can always *customise* any involved models. E.g. the `Product` model is often customised!

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.

Glossary

This is a work-in-progress list of commonly used terms when discussing Oscar.

Partner

Fulfillment partner An individual or company who can fulfil products. E.g. for physical goods, somebody with a warehouse and means of delivery.

See also:

Related model: *oscar.apps.partner.abstract_models.AbstractPartner*

Product Category Categories and subcategories are used to semantically organise your catalogue. They're merely used for navigational purposes; no business logic in Oscar considers a product's category. For instance, if you're a book shop, you could have categories such as fiction, romance, and children's books. If you'd sell both books and e-books, they could be of a different *Product Class*, but in the same category.

Product Class Product classes are an important concept in Oscar. Each product is assigned to exactly one product class. For instance, product classes could be Books, DVDs, and Toys.

Settings on a product class decide whether stock levels are *tracked* for the associated products, and whether they *require shipping*. So if you have products that require shipping and ones which don't, you'll need at least two product classes.

Used for defining *options* and *attributes* for a subset of products.

Product Options Options are values that can be associated with a item when it is added to a customer's basket. This could be something like a personalised message to be printed on a T-shirt, or a colour choice.

Product Options are different from Product Attributes in that they are used to specify a specific purchase choice by the customer, whereas Attributes exist to store and display the features of a product in a structured way.

Product Range A range is a subset of the product catalogue. It's another way of defining groups of products other than categories and product classes.

An example would be a book shop which might define a range of "Booker Prize winners". Each product will belong to different categories within the site so ranges allow them to be grouped together.

Ranges can then be used in offers (eg 10% off all booker prize winners). At some point, ranges will be expanded to have their own detail pages within Oscar too.

SKU

Stock-keeping unit. A *partner*'s way of tracking her products. Uniquely identifies a product in the partner's warehouse. Can be identical to the products *UPC*. It's stored as an attribute of *StockRecord*

See also:

Wikipedia: Stock-keeping unit

UPC

Universal Product Code A code uniquely identifying a product worldwide.

See also:

Wikipedia: Universal Product Code

Using Oscar

All you need to start developing an Oscar project.

Customising Oscar

Many parts of Oscar can be adapted to your needs like any other Django application:

- Many *settings* control Oscar's behavior
- The looks can be controlled by extending or overriding the *templates*

But as Oscar is built as a highly customisable and extendable framework, it doesn't stop there. The behaviour of all Oscar apps can heavily be altered by injecting your own code.

To extend the behavior of an Oscar core app, it needs to be forked, which is achieved with a simple management command. Afterwards, you should generally be able to override any class/model/view by just dropping it in the right place and giving it the same name.

In some cases, customising is slightly more involved. The following how-tos give plenty of examples for specific use cases:

- *How to customise models*
- *How to add views or change URLs or permissions*
- *How to customise an existing view*

For a deeper understanding of customising Oscar, the following documents are recommended:

- *Oscar design decisions*
- *Dynamic class loading*
- *Forking an app*

Fork the Oscar app

If this is the first time you're forking an Oscar app, you'll need to create a root module under which all your forked apps will live:

```
$ mkdir yourappsfolder
$ touch yourappsfolder/__init__.py
```

Now you call the helper management command which creates some basic files for you. It is explained in detail in *Forking an app*. Run it like this:

```
$ ./manage.py oscar_fork_app order yourappsfolder/
Creating folder apps/order
Creating __init__.py and admin.py
Creating models.py and copying migrations from [...] to [...]
```

Note: For forking app in project root directory, call `oscar_fork_app` with `.` (dot) instead of `yourproject/` path.

Example:

Calling `./manage.py oscar_fork_app order yourproject/ order` app will be placed in `project_root/yourproject/` directory. Calling `./manage.py oscar_fork_app order . order` app will be placed in `project_root/` directory.

Replace Oscar's app with your own in `INSTALLED_APPS`

You will need to let Django know that you replaced one of Oscar's core apps. 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 non-Oscar apps
] + get_core_apps(['yourappsfolder.order'])
```

`get_core_apps([])` will return a list of Oscar core apps. If you supply a list of additional apps, they will be used to replace the Oscar core apps. In the above example, `yourproject.order` will be returned instead of `oscar.apps.order`.

Note: Overrides of dashboard applications should follow overrides of core applications (basket, catalogue etc), since they depend on models, declared in the core applications. Otherwise, it could cause issues with Oscar's dynamic model loading.

Example:

```
INSTALLED_APPS = [
    # all your non-Oscar apps
] + get_core_apps([
    # core applications
    'yourappsfolder.catalogue',
    'yourappsfolder.order',
    # dashboard applications
    'yourappsfolder.dashboard',
    'yourappsfolder.dashboard.orders',
    'yourappsfolder.dashboard.reports',
])
```

Start customising!

You can now override every class (that is *dynamically loaded*, which is almost every class) in the app you've replaced. That means forms, views, strategies, etc. All you usually need to do is give it the same name and place it in a module with the same name.

Suppose you want to alter the way order numbers are generated. By default, the class `oscar.apps.order.utils.OrderNumberGenerator` is used. So just 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:

```
# 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
```

Dynamic class loading explained

Dynamic class loading is the foundation for making Oscar extensively customisable. It is hence worth understanding how it works, because most customisations depend on it.

It is achieved by *`oscar.core.loading.get_classes()`* and it's single-class cousin *`get_class()`*. Wherever feasible, Oscar's codebase uses `get_classes` instead of a regular import statement:

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

is replaced by:

```
from oscar.core.loading import get_class

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

Note: This is done for almost all classes: views, models, Application instances, etc. Every class imported by `get_class` can be overridden.

Why?

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

This way, most classes can be overridden with minimal duplication, as only the to-be-changed classes have to be altered. They can optionally inherit from Oscar's implementation, which often amounts to little more than a few lines of custom code for changes to core behaviour.

Seen on a bigger scale, this structures enables Oscar to ship with classes with minimal assumptions about the domain, and make it easy to modify behaviour as needed.

How it works

The `get_class` function looks through your `INSTALLED_APPS` for a matching app and will attempt to load the custom class from the specified module. If the app isn't overridden or the custom module doesn't define the class, it will fall back to the default Oscar class.

In practice

For `get_class` to pick up the customised class, the Oscar apps need to be forked. The process is detailed and illustrated with examples in *Customising Oscar*. It is usually enough to call `oscar_fork_app` and replace the app in `INSTALLED_APPS`.

Using `get_class` in your own code

Generally, there is no need for `get_class` in your own code as the location of the module for the class is known. Some Oscar developers nonetheless use `get_class` when importing classes from Oscar. This means that if someday the class is overridden, it will not require code changes. Care should be taken when doing this, as this is a tricky trade-off between maintainability and added complexity. Please note that we cannot recommend ever using `get_model` in your own code. Model initialisation is a tricky process and it's easy to run into circular import issues.

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('shipping.repository', 'Repository')
yourproject.shipping.repository.Repository  # it worked!
```

Prices and availability

This page explains how prices and availability are determined in Oscar. In short, it seems quite complicated at first as there are several parts to it, but what this buys is flexibility: buckets of it.

Overview

Simpler e-commerce frameworks often tie prices to the product model directly:

```
>>> product = Product.objects.get(id=1)
>>> product.price
Decimal('17.99')
```

Oscar, on the other hand, distinguishes products from stockrecords and provides a swappable 'strategy' component for selecting the appropriate stockrecord, calculating prices and availability information.

```
>>> from oscar.apps.partner.strategy import Selector
>>> product = Product.objects.get(id=1)
>>> strategy = Selector().strategy()
>>> info = strategy.fetch_for_product(product)

# Availability information
>>> info.availability.is_available_to_buy
```

```
True
>>> msg = info.availability.message
>>> unicode(msg)
u"In stock (58 available)"
>>> info.availability.is_purchase_permitted(59)
(False, u"A maximum of 58 can be bought")

# Price information
>>> info.price.excl_tax
Decimal('17.99')
>>> info.price.is_tax_known
True
>>> info.price.incl_tax
Decimal('21.59')
>>> info.price.tax
Decimal('3.60')
>>> info.price.currency
'GBP'
```

The product model captures the core data about the product (title, description, images) while a stockrecord represents fulfillment information for one particular partner (number in stock, base price). A product can have multiple stockrecords although only one is selected by the strategy to determine pricing and availability.

By using your own custom strategy class, a wide range of pricing, tax and availability problems can be easily solved.

The strategy class

Oscar uses a 'strategy' object to determine product availability and pricing. A new strategy instance is assigned to the request by the basket middleware. A *Selector* class determines the appropriate strategy for the request. By modifying the *Selector* class, it's possible to return different strategies for different customers.

Given a product, the strategy class is responsible for:

- Selecting a "pricing policy", an object detailing the prices of the product and whether tax is known.
- Selecting an "availability policy", an object responsible for availability logic (ie is the product available to buy) and customer messaging.
- Selecting the appropriate stockrecord to use for fulfillment. If a product can be fulfilled by several fulfilment partners, then each will have their own stockrecord.

These three entities are wrapped up in a `PurchaseInfo` object, which is a simple named tuple. The strategy class provides `fetch_for_product` and `fetch_for_parent` methods which takes a product and returns a `PurchaseInfo` instance:

The strategy class is accessed in several places in Oscar's codebase. In templates, a `purchase_info_for_product` template tag is used to load the price and availability information into the template context:

```
{% load purchase_info_tags %}
{% load currency_filters %}

{% purchase_info_for_product request product as session %}

<p>
{% if session.price.is_tax_known %}
    Price is {{ session.price.incl_tax|currency:session.price.currency }}
{% else %}
```

```
    Price is {{ session.price.excl_tax|currency:session.price.currency }} +
    tax
{% endif %}
</p>
```

Note that the `currency` template tag accepts a currency parameter from the pricing policy.

Also, basket instances have a strategy instance assigned so they can calculate prices including taxes. This is done automatically in the basket middleware.

This seems quite complicated...

While this probably seems like quite an involved way of looking up a product's price, it gives the developer an immense amount of flexibility. Here's a few examples of things you can do with a strategy class:

- Transact in multiple currencies. The strategy class can use the customer's location to select a stockrecord from a local distribution partner which will be in the local currency of the customer.
- Elegantly handle different tax models. A strategy can return prices including tax for a UK or European visitor, but without tax for US visitors where tax is only determined once shipping details are confirmed.
- Charge different prices to different customers. A strategy can return a different pricing policy depending on the user/session.
- Use a chain of preferred partners for fulfillment. A site could have many stockrecords for the same product, each from a different fulfillment partner. The strategy class could select the partner with the best margin and stock available. When stock runs out with that partner, the strategy could seamlessly switch to the next best partner.

These are the kinds of problems that other e-commerce frameworks would struggle with.

API

All strategies subclass a common `Base` class:

class `oscar.apps.partner.strategy.`**`Base`**(*request=None*)
The base strategy class

Given a product, strategies are responsible for returning a `PurchaseInfo` instance which contains:

•The appropriate stockrecord for this customer

•A pricing policy instance

•An availability policy instance

`fetch_for_line`(*line*, *stockrecord=None*)
Given a basket line instance, fetch a `PurchaseInfo` instance.

This method is provided to allow purchase info to be determined using a basket line's attributes. For instance, "bundle" products often use basket line attributes to store SKUs of contained products. For such products, we need to look at the availability of each contained product to determine overall availability.

`fetch_for_parent`(*product*)
Given a parent product, fetch a `StockInfo` instance

`fetch_for_product`(*product*, *stockrecord=None*)
Given a product, return a `PurchaseInfo` instance.

The `PurchaseInfo` class is a named tuple with attributes:

- `price`: a pricing policy object.
- `availability`: an availability policy object.
- `stockrecord`: the stockrecord that is being used

If a stockrecord is passed, return the appropriate `PurchaseInfo` instance for that product and stockrecord is returned.

Oscar also provides a "structured" strategy class which provides overridable methods for selecting the stockrecord, and determining pricing and availability policies:

class `oscar.apps.partner.strategy.`**`Structured`**(*request=None*)

A strategy class which provides separate, overridable methods for determining the 3 things that a `PurchaseInfo` instance requires:

1. A stockrecord
2. A pricing policy
3. An availability policy

`availability_policy`(*product*, *stockrecord*)

Return the appropriate availability policy

`fetch_for_product`(*product*, *stockrecord=None*)

Return the appropriate `PurchaseInfo` instance.

This method is not intended to be overridden.

`pricing_policy`(*product*, *stockrecord*)

Return the appropriate pricing policy

`select_children_stockrecords`(*product*)

Select appropriate stock record for all children of a product

`select_stockrecord`(*product*)

Select the appropriate stockrecord

For most projects, subclassing and overriding the `Structured` base class should be sufficient. However, Oscar also provides mixins to easily compose the appropriate strategy class for your domain.

Loading a strategy

Strategy instances are determined by the `Selector` class:

class `oscar.apps.partner.strategy.`**`Selector`**

Responsible for returning the appropriate strategy class for a given user/session.

This can be called in three ways:

1. Passing a request and user. This is for determining prices/availability for a normal user browsing the site.
2. Passing just the user. This is for offline processes that don't have a request instance but do know which user to determine prices for.
3. Passing nothing. This is for offline processes that don't correspond to a specific user. Eg, determining a price to store in a search index.

`strategy`(*request=None*, *user=None*, ***kwargs*)

Return an instanticated strategy instance

It's common to override this class so a custom strategy class can be returned.

Pricing policies

A pricing policy is a simple class with several properties Its job is to contain all price and tax information about a product.

There is a base class that defines the interface a pricing policy should have:

class `oscar.apps.partner.prices.`**Base**
The interface that any pricing policy must support

currency = None
Price currency (3 char code)

excl_tax = None
Price excluding tax

exists = False
Whether any prices exist

incl_tax = None
Price including tax

is_tax_known = False
Whether tax is known

retail = None
Retail price

tax = None
Price tax

There are also several policies that accommodate common scenarios:

class `oscar.apps.partner.prices.`**Unavailable**
This should be used as a pricing policy when a product is unavailable and no prices are known.

class `oscar.apps.partner.prices.`**FixedPrice**(*currency*, *excl_tax*, *tax=None*)
This should be used for when the price of a product is known in advance.

It can work for when tax isn't known (like in the US).

Note that this price class uses the tax-exclusive price for offers, even if the tax is known. This may not be what you want. Use the TaxInclusiveFixedPrice class if you want offers to use tax-inclusive prices.

Availability policies

Like pricing policies, availability policies are simple classes with several properties and methods. The job of an availability policy is to provide availability messaging to show to the customer as well as methods to determine if the product is available to buy.

The base class defines the interface:

class `oscar.apps.partner.availability.`**Base**
Base availability policy.

code = ''
Availability code. This is used for HTML classes

dispatch_date = None
When this item should be dispatched

is_available_to_buy
Test if this product is available to be bought. This is used for validation when a product is added to a user's basket.

is_purchase_permitted(*quantity*)
Test whether a proposed purchase is allowed

Should return a boolean and a reason

message = ''
A description of the availability of a product. This is shown on the product detail page. Eg "In stock", "Out of stock" etc

short_message
A shorter version of the availability message, suitable for showing on browsing pages.

There are also several pre-defined availability policies:

class oscar.apps.partner.availability.**Unavailable**
Policy for when a product is unavailable

class oscar.apps.partner.availability.**Available**
For when a product is always available, irrespective of stock level.

This might be appropriate for digital products where stock doesn't need to be tracked and the product is always available to buy.

class oscar.apps.partner.availability.**StockRequired**(*num_available*)
Allow a product to be bought while there is stock. This policy is instantiated with a stock number (num_available). It ensures that the product is only available to buy while there is stock available.

This is suitable for physical products where back orders (eg allowing purchases when there isn't stock available) are not permitted.

Strategy mixins

Oscar also ships with several mixins which implement one method of the Structured strategy. These allow strategies to be easily composed from re-usable parts:

class oscar.apps.partner.strategy.**UseFirstStockRecord**
Stockrecord selection mixin for use with the Structured base strategy. This mixin picks the first (normally only) stockrecord to fulfil a product.

This is backwards compatible with Oscar<0.6 where only one stockrecord per product was permitted.

class oscar.apps.partner.strategy.**StockRequired**
Availability policy mixin for use with the Structured base strategy. This mixin ensures that a product can only be bought if it has stock available (if stock is being tracked).

class oscar.apps.partner.strategy.**NoTax**
Pricing policy mixin for use with the Structured base strategy. This mixin specifies zero tax and uses the price_excl_tax from the stockrecord.

class oscar.apps.partner.strategy.**FixedRateTax**
Pricing policy mixin for use with the Structured base strategy. This mixin applies a fixed rate tax to the base price from the product's stockrecord. The price_incl_tax is quantized to two decimal places. Rounding behaviour is Decimal's default

get_exponent(*stockrecord*)
This method serves as hook to be able to plug in support for a varying exponent based on the currency.

TODO: Needs tests.

get_rate(*product*, *stockrecord*)
This method serves as hook to be able to plug in support for varying tax rates based on the product.

TODO: Needs tests.

class oscar.apps.partner.strategy.**DeferredTax**
Pricing policy mixin for use with the Structured base strategy. This mixin does not specify the product tax and is suitable to territories where tax isn't known until late in the checkout process.

Default strategy

Oscar's default Selector class returns a Default strategy built from the strategy mixins:

```
class Default(UseFirstStockRecord, StockRequired, NoTax, Structured):
    pass
```

The behaviour of this strategy is:

- Always picks the first stockrecord (this is backwards compatible with Oscar<0.6 where a product could only have one stockrecord).
- Charge no tax.
- Only allow purchases where there is appropriate stock (eg no back-orders).

How to use

There's lots of ways to use strategies, pricing and availability policies to handle your domain's requirements.

The normal first step is provide your own Selector class which returns a custom strategy class. Your custom strategy class can be composed of the above mixins or your own custom logic.

Example 1: UK VAT

Here's an example strategy.py module which is used to charge VAT on prices.

```
# myproject/partner/strategy.py

from oscar.apps.partner import strategy, prices


class Selector(object):
    """
    Custom selector to return a UK-specific strategy that charges VAT
    """

    def strategy(self, request=None, user=None, **kwargs):
        return UKStrategy()


class IncludingVAT(strategy.FixedRateTax):
    """
    Price policy to charge VAT on the base price
    """
    # We can simply override the tax rate on the core FixedRateTax.  Note
    # this is a simplification: in reality, you might want to store tax
    # rates and the date ranges they apply in a database table.  Your
```

```
    # pricing policy could simply look up the appropriate rate.
    rate = D('0.20')


class UKStrategy(strategy.UseFirstStockRecord, IncludingVAT,
                 strategy.StockRequired, strategy.Structured):
    """
    Typical UK strategy for physical goods.

    - There's only one warehouse/partner so we use the first and only stockrecord
    - Enforce stock level.  Don't allow purchases when we don't have stock.
    - Charge UK VAT on prices.  Assume everything is standard-rated.
    """
```

Example 2: US sales tax

Here's an example `strategy.py` module which is suitable for use in the US where taxes can't be calculated until the shipping address is known. You normally need to use a 3rd party service to determine taxes - details omitted here.

```
from oscar.apps.partner import strategy, prices


class Selector(object):
    """
    Custom selector class to returns a US strategy
    """

    def strategy(self, request=None, user=None, **kwargs):
        return USStrategy()


class USStrategy(strategy.UseFirstStockRecord, strategy.DeferredTax,
                 strategy.StockRequired, strategy.Structured):
    """
    Typical US strategy for physical goods.  Note we use the ``DeferredTax``
    mixin to ensure prices are returned without tax.

    - Use first stockrecord
    - Enforce stock level
    - Taxes aren't known for prices at this stage
    """
```

Deploying Oscar

Oscar is a just a set of Django apps - it doesn't have any special deployment requirements. That means the excellent Django docs for deployment should be your first stop. This page then only distills some of the experience gained from running Oscar projects.

Performance

Setting up caching is crucial for a good performance. Oscar's templates are split into many partials, hence it is recommended to use the cached template loader. Sorl also will hit your database hard if you run it without a cache backend.

If your memory constraints are tight and you can only run one Python worker, LocMemCache will usually outperform external cache backends due to the lower overhead. But once you can scale beyond one worker, it might make good sense to switch to something like memcached or redis.

Blocking in views should be avoided if possible. That is especially true for external API calls and sending emails. Django's pluggable email backends allow for switching out the blocking SMTP backend to a custom non-blocking solution. Possible options are storing emails in a database or cache for later consumption or triggering an external worker, e.g. via django-celery. django_post-office works nicely.

For backwards-compatibility reasons, Django doesn't enable database connection pooling by default. Performance is likely to improve when enabled.

Security

Oscar relies on the Django framework for security measures and therefore no Oscar specific configurations with regard to security are in place. See Django's guidelines for security for more information.

django-secure is a nice app that comes with a few sanity checks for deployments behind SSL.

Search Engine Optimisation

A basic example of what a sitemap for Oscar could look like has been added to the sandbox site. Have a look at `sandbox/urls.py` for inspiration.

Translation

All Oscar translation work is done on Transifex. If you'd like to contribute, just apply for a language and go ahead! The source strings in Transifex are updated after every commit on Oscar's master branch on GitHub. We only pull the translation strings back into Oscar's repository when preparing for a release. That means your most recent translations will always be on Transifex, not in the repo!

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.

Upgrading

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

Migrations

Oscar provides 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 makemigrations shipping
```

to migrate your shipping app.

Migrating customised apps (models unchanged)

If you have customised an app, but have not touched the models nor migrations, you're best off copying the migrations from Oscar. This approach has the advantage of pulling in any data migrations. Find the updated(!) Oscar in your virtualenv or clone the Oscar repo at the correct version tag. Then find the migrations, copy them across, and migrate as usual. You will have to adapt paths, but something akin to this will work:

```
$ cdsitepackages oscar/apps/shipping/migrations
$ copy *.py <your_project>/myshop/shipping/migrations/
```

Migrating customised apps (models changed)

At this point, you have essentially forked away from Oscar's migrations. Read the release notes carefully and see if it includes data migrations. If not, it's as easy as:

```
./manage.py makemigrations shipping
```

to create the appropriate migration.

But if there is data migrations, you will need to look into what they do, and likely will have to imitate what they're doing. You can copy across the data migration, but you have to manually update the dependencies.

If there's no schema migrations, you should set the data migration to depend on your last migration for that app. If there is a schema migration, you will have to imitate the dependency order of Oscar.

Feel free to get in touch on the mailing list if you run into any problems.

Forking an app

This guide explains how to fork an app in Oscar.

Note: The following steps are now automated by the `oscar_fork_app` management command. They're explained in detail so you get an idea of what's going on. But there's no need to do this manually anymore! More information is available in *Fork the Oscar app*.

Create Python module with same label

You need to create a Python module with the same "app label" as the Oscar app you want to extend. E.g., to create a local version of `oscar.apps.order`, do the following:

```
$ mkdir yourproject/order
$ touch yourproject/order/__init__.py
```

Reference Oscar's models

If the original Oscar app has a `models.py`, you'll need to create a `models.py` file in your local app. It should import all models from the Oscar app being overridden:

```
# yourproject/order/models.py

# your custom models go here

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

If two models with the same name are declared within an app, Django will only use the first one. That means that if you wish to customise Oscar's models, you must declare your custom ones before importing Oscar's models for that app.

You have to copy the `migrations` directory from `oscar/apps/order` and put it into your `order` app. Detailed instructions are available in *How to customise models*.

Get the Django admin working

When you replace one of Oscar's apps with a local one, Django admin integration is lost. If you'd like to use it, you need to create an `admin.py` and import the core app's `admin.py` (which will run the register code):

```
# yourproject/order/admin.py
import oscar.apps.order.admin
```

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

Use supplied app config

Oscar ships with an app config for each app, which sets app labels and runs startup code. You need to make sure that happens.

Reference:

Core functionality

This page details the core classes and functions that Oscar uses. These aren't specific to one particular app, but are used throughout Oscar's codebase.

Dynamic class loading

The key to Oscar's flexibility is dynamically loading classes. This allows projects to provide their own versions of classes that extend and override the core functionality.

`oscar.core.loading.`**`get_classes`**(*module_label*, *classnames*, *module_prefix='oscar.apps'*)

Dynamically import a list of classes from the given module.

This works by looping over `INSTALLED_APPS` and looking for a match against the passed module label. If the requested class can't be found in the matching module, then we attempt to import it from the corresponding core app.

This is very similar to `django.db.models.get_model` function for dynamically loading models. This function is more general though as it can load any class from the matching app, not just a model.

Parameters

- **`module_label`** (*str*) – Module label comprising the app label and the module name, separated by a dot. For example, 'catalogue.forms'.
- **`classname`** (*str*) – Name of the class to be imported.

Returns The requested class object or `None` if it can't be found

Examples

Load a single class:

```
>>> get_class('dashboard.catalogue.forms', 'ProductForm')
oscar.apps.dashboard.catalogue.forms.ProductForm
```

Load a list of classes:

```
>>> get_classes('dashboard.catalogue.forms',
...             ['ProductForm', 'StockRecordForm'])
[oscar.apps.dashboard.catalogue.forms.ProductForm,
 oscar.apps.dashboard.catalogue.forms.StockRecordForm]
```

Raises

- `AppNotFoundError` – If no app is found in `INSTALLED_APPS` that matches the passed module label.
- `ImportError` – If the attempted import of a class raises an `ImportError`, it is re-raised

`oscar.core.loading.`**`get_class`**(*module_label*, *classname*, *module_prefix='oscar.apps'*)

Dynamically import a single class from the given module.

This is a simple wrapper around *get_classes* for the case of loading a single class.

Parameters

- **`module_label`** (*str*) – Module label comprising the app label and the module name, separated by a dot. For example, 'catalogue.forms'.
- **`classname`** (*str*) – Name of the class to be imported.

Returns The requested class object or *None* if it can't be found

URL patterns and views

Oscar's app organise their URLs and associated views using a "Application" class instance. This works in a similar way to Django's admin app, and allows Oscar projects to subclass and customised URLs and views.

class `oscar.core.application.`**`Application`**(*app_name=None*, ***kwargs*)
Base application class.

This is subclassed by each app to provide a customisable container for an app's views and permissions.

`default_permissions` = None
Default permission for any view not in permissions_map

`get_permissions`(*url*)
Return a list of permissions for a given URL name

Parameters **`url`** (*str*) – A URL name (eg `basket.basket`)

Returns A list of permission strings.

Return type list

`get_url_decorator`(*pattern*)
Return the appropriate decorator for the view function with the passed URL name. Mainly used for access-protecting views.

It's possible to specify:

•no permissions necessary: use None

•a set of permissions: use a list

•two set of permissions (*or*): use a two-tuple of lists

See permissions_required decorator for details

`get_urls`()
Return the url patterns for this app.

`hidable_feature_name` = None
A name that allows the functionality within this app to be disabled

`name` = None
Namespace name

`permissions_map` = {}
Maps view names to lists of permissions. We expect tuples of lists as dictionary values. A list is a set of permissions that all needto be fulfilled (AND). Only one set of permissions has to be fulfilled (OR). If there's only one set of permissions, as a shortcut, you can also just define one list.

`post_process_urls`(*urlpatterns*)
Customise URL patterns.

This method allows decorators to be wrapped around an apps URL patterns.

By default, this only allows custom decorators to be specified, but you could override this method to do anything you want.

Parameters **`urlpatterns`** (*list*) – A list of URL patterns

Prices

Oscar uses a custom price object for easier tax handling.

class oscar.core.prices.**Price**(*currency*, *excl_tax*, *incl_tax=None*, *tax=None*)
Simple price class that encapsulates a price and its tax information

incl_tax
Decimal – Price including taxes

excl_tax
Decimal – Price excluding taxes

tax
Decimal – Tax amount

is_tax_known
bool – Whether tax is known

currency
str – 3 character currency code

Custom model fields

Oscar uses a few custom model fields.

class oscar.models.fields.**Creator**(*field*)
A placeholder class that provides a way to set the attribute on the model.

class oscar.models.fields.**NullCharField**(**args*, ***kwargs*)
CharField that stores '' as None and returns None as '' Useful when using unique=True and forms. Implies null==blank==True.

When a ModelForm with a CharField with null=True gets saved, the field will be set to '': https://code.djangoproject.com/ticket/9590 This breaks usage with unique=True, as '' is considered equal to another field set to ''.

deconstruct()
deconstruct() is needed by Django's migration framework

class oscar.models.fields.**PositiveDecimalField**(*verbose_name=None*, *name=None*, *max_digits=None*, *decimal_places=None*, ***kwargs*)
A simple subclass of django.db.models.fields.DecimalField that restricts values to be non-negative.

class oscar.models.fields.**UppercaseCharField**(**args*, ***kwargs*)
A simple subclass of django.db.models.fields.CharField that restricts all text to be uppercase.

Defined with the with_metaclass helper so that to_python is called https://docs.djangoproject.com/en/1.6/howto/custom-model-fields/#the-subfieldbase-metaclass # NOQA

Oscar Core Apps explained

Oscar is split up in multiple, mostly independent apps.

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.

active_address_fields(*include_salutation=True*)
Return the non-empty components of the address, but merging the title, first_name and last_name into a single line.

ensure_postcode_is_valid_for_country()
Validate postcode given the country

generate_hash()
Returns a hash of the address summary

join_fields(*fields*, *separator=u', '*)
Join a sequence of fields using the specified separator

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
Name (including title)

search_text
A field only used for searching addresses - this contains all the relevant fields. This is effectively a poor man's Solr text field.

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

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

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

The field names are a bit awkward, but kept for backwards compatibility. pycountry's syntax of alpha2, alpha3, name and official_name seems sane.

code
Shorthand for the ISO 3166 Alpha-2 code

name
The full official name of a country e.g. 'United Kingdom of Great Britain and Northern Ireland'

numeric_code
Shorthand for the ISO 3166 numeric code.

iso_3166_1_numeric used to wrongly be a integer field, but has to be padded with leading zeroes. It's since been converted to a char field, but the database might still contain non-padded strings. That's why the padding is kept.

printable_name
The commonly used name; e.g. 'United Kingdom'

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

A partner can have one or more addresses. This can be useful e.g. when determining US tax which depends on the origin of the shipment.

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.

NOTE: ShippingAddress is a model of the order app. But moving it there is tricky due to circular import issues that are amplified by get_model/get_class calls pre-Django 1.7 to register receivers. So... TODO: Once Django 1.6 support is dropped, move AbstractBillingAddress and AbstractShippingAddress to the order app, and move PartnerAddress to the partner app.

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.

hash
A hash is kept to try and avoid duplicate addresses being added to the address book.

is_default_for_billing
Whether this address should be the default for billing.

is_default_for_shipping
Whether this address is the default for shipping

num_orders_as_billing_address
Same as previous, but for billing address.

num_orders_as_shipping_address
We keep track of the number of times an address has been used as a shipping address so we can show the most popular ones first at the checkout.

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

Views

None.

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.

Abstract models

class oscar.apps.analytics.abstract_models.**AbstractProductRecord**(**args*, ***kwargs*)

A record of a how popular a product is.

This used be auto-merchandising to display the most popular products.

class oscar.apps.analytics.abstract_models.**AbstractUserRecord**(**args*, ***kwargs*)

A record of a user's activity.

Views

None.

Basket

The basket app handles shopping baskets, which essentially are a collection of products that hopefully end up being ordered.

Abstract models

class oscar.apps.basket.abstract_models.**AbstractBasket**(**args*, ***kwargs*)

Basket object

add(*product*, *quantity=1*, *options=None*)

Add a product to the basket

'stock_info' is the price and availability data returned from a partner strategy class.

The 'options' list should contains dicts with keys 'option' and 'value' which link the relevant product.Option model and string value respectively.

Returns (line, created). line: the matching basket line created: whether the line was created or updated

add_product(*product*, *quantity=1*, *options=None*)

Add a product to the basket

'stock_info' is the price and availability data returned from a partner strategy class.

The 'options' list should contains dicts with keys 'option' and 'value' which link the relevant product.Option model and string value respectively.

Returns (line, created). line: the matching basket line created: whether the line was created or updated

all_lines()

Return a cached set of basket lines.

This is important for offers as they alter the line models and you don't want to reload them from the DB as that information would be lost.

applied_offers()

Return a dict of offers successfully applied to the basket.

This is used to compare offers before and after a basket change to see if there is a difference.

can_be_edited

Test if a basket can be edited

contains_voucher(*code*)
Test whether the basket contains a voucher with a given code

flush()
Remove all lines from basket.

freeze()
Freezes the basket so it cannot be modified.

grouped_voucher_discounts
Return discounts from vouchers but grouped so that a voucher which links to multiple offers is aggregated into one object.

is_empty
Test if this basket is empty

is_quantity_allowed(*qty*)
Test whether the passed quantity of items can be added to the basket

is_shipping_required()
Test whether the basket contains physical products that require shipping.

is_tax_known
Test if tax values are known for this basket

line_quantity(*product*, *stockrecord*, *options=None*)
Return the current quantity of a specific product and options

merge(*basket*, *add_quantities=True*)
Merges another basket with this one.

Basket The basket to merge into this one.

Add_quantities Whether to add line quantities when they are merged.

merge_line(*line*, *add_quantities=True*)
For transferring a line from another basket to this one.

This is used with the "Saved" basket functionality.

num_items
Return number of items

num_lines
Return number of lines

offer_discounts
Return basket discounts from non-voucher sources. Does not include shipping discounts.

post_order_actions
Return discounts from vouchers

product_quantity(*product*)
Return the quantity of a product in the basket

The basket can contain multiple lines with the same product, but different options and stockrecords. Those quantities are summed up.

reset_offer_applications()
Remove any discounts so they get recalculated

set_as_submitted()
Mark this basket as submitted

shipping_discounts
Return discounts from vouchers

submit()
Mark this basket as submitted

thaw()
Unfreezes a basket so it can be modified again

total_excl_tax
Return total line price excluding tax

total_excl_tax_excl_discounts
Return total price excluding tax and discounts

total_incl_tax
Return total price inclusive of tax and discounts

total_incl_tax_excl_discounts
Return total price inclusive of tax but exclusive discounts

total_tax
Return total tax for a line

voucher_discounts
Return discounts from vouchers

class oscar.apps.basket.abstract_models.**AbstractLine**(**args*, ***kwargs*)
A line of a basket (product and a quantity)

Common approaches on ordering basket lines:

1. First added at top. That's the history-like approach; new items are added to the bottom of the list. Changing quantities doesn't impact position. Oscar does this by default. It just sorts by Line.pk, which is guaranteed to increment after each creation.

2. Last modified at top. That means items move to the top when you add another one, and new items are added to the top as well. Amazon mostly does this, but doesn't change the position when you update the quantity in the basket view. To get this behaviour, add a date_updated field, change Meta.ordering and optionally do something similar on wishlist lines. Order lines should already be created in the order of the basket lines, and are sorted by their primary key, so no changes should be necessary there.

clear_discount()
Remove any discounts from this line.

consume(*quantity*)
Mark all or part of the line as 'consumed'

Consumed items are no longer available to be used in offers.

discount(*discount_value*, *affected_quantity*, *incl_tax=True*)
Apply a discount to this line

get_price_breakdown()
Return a breakdown of line prices after discounts have been applied.

Returns a list of (unit_price_incl_tax, unit_price_excl_tax, quantity) tuples.

get_warning()
Return a warning message about this basket line if one is applicable

This could be things like the price has changed

purchase_info
Return the stock/price info

unit_effective_price
The price to use for offer calculations

class oscar.apps.basket.abstract_models.**AbstractLineAttribute**(**args*, ***kwargs*)
An attribute of a basket line

Views

class oscar.apps.basket.views.**BasketAddView**(***kwargs*)
Handles the add-to-basket submissions, which are triggered from various parts of the site. The add-to-basket form is loaded into templates using a templatetag from module basket_tags.py.

product_model
alias of Product

Catalogue

This is an essential Oscar app which exposes functionality to manage your product catalogue. *oscar.apps.catalogue.abstract_models.AbstractProduct* is its main model. The catalogue app also includes views specific to viewing a list or individual products.

Abstract models

class oscar.apps.catalogue.abstract_models.**AbstractAttributeOption**(**args*, ***kwargs*)
Provides an option within an option group for an attribute type Examples: In a Language group, English, Greek, French

class oscar.apps.catalogue.abstract_models.**AbstractAttributeOptionGroup**(**args*, ***kwargs*)
Defines a group of options that collectively may be used as an attribute type

For example, Language

class oscar.apps.catalogue.abstract_models.**AbstractCategory**(**args*, ***kwargs*)
A product category. Merely used for navigational purposes; has no effects on business logic.

Uses django-treebeard.

ensure_slug_uniqueness()
Ensures that the category's slug is unique amongst it's siblings. This is inefficient and probably not thread-safe.

full_name
Returns a string representation of the category and it's ancestors, e.g. 'Books > Non-fiction > Essential programming'.

It's rarely used in Oscar's codebase, but used to be stored as a CharField and is hence kept for backwards compatibility. It's also sufficiently useful to keep around.

full_slug
Returns a string of this category's slug concatenated with the slugs of it's ancestors, e.g. 'books/non-fiction/essential-programming'.

Oscar used to store this as in the 'slug' model field, but this field has been re-purposed to only store this category's slug and to not include it's ancestors' slugs.

generate_slug()
Generates a slug for a category. This makes no attempt at generating a unique slug.

get_absolute_url()
Our URL scheme means we have to look up the category's ancestors. As that is a bit more expensive, we cache the generated URL. That is safe even for a stale cache, as the default implementation of ProductCategoryView does the lookup via primary key anyway. But if you change that logic, you'll have to reconsider the caching approach.

get_ancestors_and_self()
Gets ancestors and includes itself. Use treebeard's get_ancestors if you don't want to include the category itself. It's a separate function as it's commonly used in templates.

get_descendants_and_self()
Gets descendants and includes itself. Use treebeard's get_descendants if you don't want to include the category itself. It's a separate function as it's commonly used in templates.

save(**args*, ***kwargs*)
Oscar traditionally auto-generated slugs from names. As that is often convenient, we still do so if a slug is not supplied through other means. If you want to control slug creation, just create instances with a slug already set, or expose a field on the appropriate forms.

class oscar.apps.catalogue.abstract_models.**AbstractOption**(**args*, ***kwargs*)
An option that can be selected for a particular item when the product is added to the basket.

For example, a list ID for an SMS message send, or a personalised message to print on a T-shirt.

This is not the same as an 'attribute' as options do not have a fixed value for a particular item. Instead, option need to be specified by a customer when they add the item to their basket.

class oscar.apps.catalogue.abstract_models.**AbstractProduct**(**args*, ***kwargs*)
The base product object

There's three kinds of products; they're distinguished by the structure field.

- A stand alone product. Regular product that lives by itself.
- A child product. All child products have a parent product. They're a specific version of the parent.
- A parent product. It essentially represents a set of products.

An example could be a yoga course, which is a parent product. The different times/locations of the courses would be associated with the child products.

attribute_summary
Return a string of all of a product's attributes

calculate_rating()
Calculate rating value

can_be_parent(*give_reason=False*)
Helps decide if a the product can be turned into a parent product.

clean()
Validate a product. Those are the rules:

	stand alone	parent	child
title	required	required	optional
product class	required	required	must be None
parent	forbidden	forbidden	required
stockrecords	0 or more	forbidden	0 or more
categories	1 or more	1 or more	forbidden
attributes	optional	optional	optional
rec. products	optional	optional	unsupported
options	optional	optional	forbidden

Because the validation logic is quite complex, validation is delegated to the sub method appropriate for the product's structure.

get_absolute_url()
Return a product's absolute url

get_categories()
Return a product's categories or parent's if there is a parent product.

get_is_discountable()
At the moment, is_discountable can't be set individually for child products; they inherit it from their parent.

get_missing_image()
Returns a missing image object.

get_product_class()
Return a product's item class. Child products inherit their parent's.

get_title()
Return a product's title or it's parent's title if it has no title

has_stockrecords
Test if this product has any stockrecords

is_discountable
Determines if a product may be used in an offer. It is illegal to discount some types of product (e.g. ebooks) and this field helps merchants from avoiding discounting such products Note that this flag is ignored for child products; they inherit from the parent product.

is_review_permitted(*user*)
Determines whether a user may add a review on this product.

Default implementation respects OSCAR_ALLOW_ANON_REVIEWS and only allows leaving one review per user and product.

Override this if you want to alter the default behaviour; e.g. enforce that a user purchased the product to be allowed to leave a review.

options
Returns a set of all valid options for this product. It's possible to have options product class-wide, and per product.

primary_image()
Returns the primary image for a product. Usually used when one can only display one product image, e.g. in a list of products.

product_class
"Kind" of product, e.g. T-Shirt, Book, etc. None for child products, they inherit their parent's product class

product_options
It's possible to have options product class-wide, and per product.

sorted_recommended_products
Keeping order by recommendation ranking.

update_rating()
Recalculate rating field

class oscar.apps.catalogue.abstract_models.**AbstractProductAttribute**(**args*, ***kwargs*)
Defines an attribute for a product class. (For example, number_of_pages for a 'book' class)

class oscar.apps.catalogue.abstract_models.**AbstractProductAttributeValue**(**args*, ***kwargs*)
The "through" model for the m2m relationship between catalogue.Product and catalogue.ProductAttribute. This specifies the value of the attribute for a particular product

For example: number_of_pages = 295

summary()
Gets a string representation of both the attribute and it's value, used e.g in product summaries.

value_as_html
Returns a HTML representation of the attribute's value. To customise e.g. image attribute values, declare a _image_as_html property and return e.g. an <img> tag. Defaults to the _as_text representation.

value_as_text
Returns a string representation of the attribute's value. To customise e.g. image attribute values, declare a _image_as_text property and return something appropriate.

class oscar.apps.catalogue.abstract_models.**AbstractProductCategory**(**args*, ***kwargs*)
Joining model between products and categories. Exists to allow customising.

class oscar.apps.catalogue.abstract_models.**AbstractProductClass**(**args*, ***kwargs*)
Used for defining options and attributes for a subset of products. E.g. Books, DVDs and Toys. A product can only belong to one product class.

At least one product class must be created when setting up a new Oscar deployment.

Not necessarily equivalent to top-level categories but usually will be.

options
These are the options (set by the user when they add to basket) for this item class. For instance, a product class of "SMS message" would always require a message to be specified before it could be bought. Note that you can also set options on a per-product level.

requires_shipping
Some product type don't require shipping (eg digital products) - we use this field to take some shortcuts in the checkout.

track_stock
Digital products generally don't require their stock levels to be tracked.

class oscar.apps.catalogue.abstract_models.**AbstractProductImage**(**args*, ***kwargs*)
An image of a product

delete(**args*, ***kwargs*)
Always keep the display_order as consecutive integers. This avoids issue #855.

display_order
Use display_order to determine which is the "primary" image

is_primary()
Return bool if image display order is 0

class oscar.apps.catalogue.abstract_models.**AbstractProductRecommendation**(**args*, ***kwargs*)

'Through' model for product recommendations

class oscar.apps.catalogue.abstract_models.**MissingProductImage**(*name=None*)

Mimics a Django file field by having a name property.

sorl-thumbnail requires all it's images to be in MEDIA_ROOT. This class tries symlinking the default "missing image" image in STATIC_ROOT into MEDIA_ROOT for convenience, as that is necessary every time an Oscar project is setup. This avoids the less helpful NotFound IOError that would be raised when sorl-thumbnail tries to access it.

Views

class oscar.apps.catalogue.views.**CatalogueView**(***kwargs*)

Browse all products in the catalogue

class oscar.apps.catalogue.views.**ProductCategoryView**(***kwargs*)

Browse products in a given category

get_categories()

Return a list of the current category and its ancestors

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.

Abstract models

None.

Views and mixins

class oscar.apps.checkout.views.**IndexView**(***kwargs*)
First page of the checkout. We prompt user to either sign in, or to proceed as a guest (where we still collect their email address).

class oscar.apps.checkout.views.**PaymentDetailsView**(***kwargs*)
For taking the details of payment and creating the order.

This view class is used by two separate URLs: 'payment-details' and 'preview'. The *preview* class attribute is used to distinguish which is being used. Chronologically, *payment-details* (preview=False) comes before *preview* (preview=True).

If sensitive details are required (eg a bankcard), then the payment details view should submit to the preview URL and a custom implementation of *validate_payment_submission* should be provided.

- If the form data is valid, then the preview template can be rendered with the payment-details forms re-rendered within a hidden div so they can be re-submitted when the 'place order' button is clicked. This avoids having to write sensitive data to disk anywhere during the process. This can be done by calling *render_preview*, passing in the extra template context vars.
- If the form data is invalid, then the payment details templates needs to be re-rendered with the relevant error messages. This can be done by calling *render_payment_details*, passing in the form instances to pass to the templates.

The class is deliberately split into fine-grained methods, responsible for only one thing. This is to make it easier to subclass and override just one component of functionality.

All projects will need to subclass and customise this class as no payment is taken by default.

get_default_billing_address()
Return default billing address for user

This is useful when the payment details view includes a billing address form - you can use this helper method to prepopulate the form.

Note, this isn't used in core oscar as there is no billing address form by default.

handle_payment_details_submission(*request*)
Handle a request to submit payment details.

This method will need to be overridden by projects that require forms to be submitted on the payment details view. The new version of this method should validate the submitted form data and:

- If the form data is valid, show the preview view with the forms re-rendered in the page
- If the form data is invalid, show the payment details view with the form errors showing.

handle_place_order_submission(*request*)
Handle a request to place an order.

This method is normally called after the customer has clicked "place order" on the preview page. It's responsible for (re-)validating any form information then building the submission dict to pass to the *submit* method.

If forms are submitted on your payment details view, you should override this method to ensure they are valid before extracting their data into the submission dict and passing it onto *submit*.

render_payment_details(*request*, ***kwargs*)
Show the payment details page

This method is useful if the submission from the payment details view is invalid and needs to be re-rendered with form errors showing.

render_preview(*request*, ***kwargs*)
Show a preview of the order.

If sensitive data was submitted on the payment details page, you will need to pass it back to the view here so it can be stored in hidden form inputs. This avoids ever writing the sensitive data to disk.

submit(*user*, *basket*, *shipping_address*, *shipping_method*, *shipping_charge*, *billing_address*, *order_total*, *payment_kwargs=None*, *order_kwargs=None*)
Submit a basket for order placement.

The process runs as follows:

•Generate an order number

•Freeze the basket so it cannot be modified any more (important when redirecting the user to another site for payment as it prevents the basket being manipulated during the payment process).

•Attempt to take payment for the order - If payment is successful, place the order - If a redirect is required (eg PayPal, 3DSecure), redirect - If payment is unsuccessful, show an appropriate error message

Basket The basket to submit.

Payment_kwargs Additional kwargs to pass to the handle_payment method. It normally makes sense to pass form instances (rather than model instances) so that the forms can be re-rendered correctly if payment fails.

Order_kwargs Additional kwargs to pass to the place_order method

class oscar.apps.checkout.views.**PaymentMethodView**(***kwargs*)
View for a user to choose which payment method(s) they want to use.

This would include setting allocations if payment is to be split between multiple sources. It's not the place for entering sensitive details like bankcard numbers though - that belongs on the payment details view.

class oscar.apps.checkout.views.**ShippingAddressView**(***kwargs*)
Determine the shipping address for the order.

The default behaviour is to display a list of addresses from the users's address book, from which the user can choose one to be their shipping address. They can add/edit/delete these USER addresses. This address will be automatically converted into a SHIPPING address when the user checks out.

Alternatively, the user can enter a SHIPPING address directly which will be saved in the session and later saved as ShippingAddress model when the order is successfully submitted.

class oscar.apps.checkout.views.**ShippingMethodView**(***kwargs*)
View for allowing a user to choose a shipping method.

Shipping methods are largely domain-specific and so this view will commonly need to be subclassed and customised.

The default behaviour is to load all the available shipping methods using the shipping Repository. If there is only 1, then it is automatically selected. Otherwise, a page is rendered where the user can choose the appropriate one.

get_available_shipping_methods()
Returns all applicable shipping method objects for a given basket.

class oscar.apps.checkout.views.**ThankYouView**(***kwargs*)
Displays the 'thank you' page which summarises the order just submitted.

class oscar.apps.checkout.views.**UserAddressDeleteView**(***kwargs*)
Delete an address from a user's address book.

class oscar.apps.checkout.views.**UserAddressUpdateView**(***kwargs*)
Update a user address

class oscar.apps.checkout.mixins.**OrderPlacementMixin**
Mixin which provides functionality for placing orders.

Any view class which needs to place an order should use this mixin.

add_payment_event(*event_type_name*, *amount*, *reference=''*)
Record a payment event for creation once the order is placed

add_payment_source(*source*)
Record a payment source for this order

create_billing_address(*user*, *billing_address=None*, *shipping_address=None*, ***kwargs*)
Saves any relevant billing data (eg a billing address).

create_shipping_address(*user*, *shipping_address*)
Create and return the shipping address for the current order.

Compared to self.get_shipping_address(), ShippingAddress is saved and makes sure that appropriate UserAddress exists.

freeze_basket(*basket*)
Freeze the basket so it can no longer be modified

generate_order_number(*basket*)
Return a new order number

handle_order_placement(*order_number*, *user*, *basket*, *shipping_address*, *shipping_method*, *shipping_charge*, *billing_address*, *order_total*, ***kwargs*)
Write out the order models and return the appropriate HTTP response

We deliberately pass the basket in here as the one tied to the request isn't necessarily the correct one to use in placing the order. This can happen when a basket gets frozen.

handle_payment(*order_number*, *total*, ***kwargs*)
Handle any payment processing and record payment sources and events.

This method is designed to be overridden within your project. The default is to do nothing as payment is domain-specific.

This method is responsible for handling payment and recording the payment sources (using the add_payment_source method) and payment events (using add_payment_event) so they can be linked to the order when it is saved later on.

handle_successful_order(*order*)
Handle the various steps required after an order has been successfully placed.

Override this view if you want to perform custom actions when an order is submitted.

place_order(*order_number*, *user*, *basket*, *shipping_address*, *shipping_method*, *shipping_charge*, *order_total*, *billing_address=None*, ***kwargs*)
Writes the order out to the DB including the payment models

restore_frozen_basket()
Restores a frozen basket as the sole OPEN basket. Note that this also merges in any new products that have been added to a basket that has been created while payment.

save_payment_details(*order*)
Saves all payment-related details. This could include a billing address, payment sources and any order payment events.

save_payment_events(*order*)
Saves any relevant payment events for this order

save_payment_sources(*order*)
Saves any payment sources used in this order.

When the payment sources are created, the order model does not exist and so they need to have it set before saving.

update_address_book(*user*, *addr*)
Update the user's address book based on the new shipping address

class oscar.apps.checkout.session.**CheckoutSessionMixin**
Mixin to provide common functionality shared between checkout views.

All checkout views subclass this mixin. It ensures that all relevant checkout information is available in the template context.

build_submission(***kwargs*)
Return a dict of data that contains everything required for an order submission. This includes payment details (if any).

This can be the right place to perform tax lookups and apply them to the basket.

check_basket_is_valid(*request*)
Check that the basket is permitted to be submitted as an order. That is, all the basket lines are available to buy - nothing has gone out of stock since it was added to the basket.

get_billing_address(*shipping_address*)
Return an unsaved instance of the billing address (if one exists)

This method only returns a billing address if the session has been used to store billing address information. It's also possible to capture billing address information as part of the payment details forms, which never get stored in the session. In that circumstance, the billing address can be set directly in the build_submission dict.

get_order_totals(*basket*, *shipping_charge*, ***kwargs*)
Returns the total for the order with and without tax

get_pre_conditions(*request*)
Return the pre-condition method names to run for this view

get_shipping_address(*basket*)
Return the (unsaved) shipping address for this checkout session.

If the shipping address was entered manually, then we instantiate a `ShippingAddress` model with the appropriate form data (which is saved in the session).

If the shipping address was selected from the user's address book, then we convert the `UserAddress` to a `ShippingAddress`.

The `ShippingAddress` instance is not saved as sometimes you need a shipping address instance before the order is placed. For example, if you are submitting fraud information as part of a payment request.

The `OrderPlacementMixin.create_shipping_address` method is responsible for saving a shipping address when an order is placed.

get_shipping_method(*basket*, *shipping_address=None*, ***kwargs*)
Return the selected shipping method instance from this checkout session

The shipping address is passed as we need to check that the method stored in the session is still valid for the shipping address.

get_skip_conditions(*request*)
Return the skip-condition method names to run for this view

Forms

Utils

class oscar.apps.checkout.calculators.**OrderTotalCalculator**(*request=None*)
Calculator class for calculating the order total.

class oscar.apps.checkout.utils.**CheckoutSessionData**(*request*)
Responsible for marshalling all the checkout session data

Multi-stage checkouts often require several forms to be submitted and their data persisted until the final order is placed. This class helps store and organise checkout form data until it is required to write out the final order.

bill_to_new_address(*address_fields*)
Store address fields for a billing address.

bill_to_shipping_address()
Record fact that the billing address is to be the same as the shipping address.

bill_to_user_address(*address*)
Set an address from a user's address book as the billing address

Address The address object

billing_address_same_as_shipping()
Record fact that the billing address is to be the same as the shipping address.

billing_user_address_id()
Return the ID of the user address being used for billing

flush()
Flush all session data

is_billing_address_set()
Test whether a billing address has been stored in the session.

This can be from a new address or re-using an existing address.

is_shipping_address_set()
Test whether a shipping address has been stored in the session.

This can be from a new address or re-using an existing address.

is_shipping_method_set(*basket*)
Test if a valid shipping method is stored in the session

new_billing_address_fields()
Return fields for a billing address

new_shipping_address_fields()
Return shipping address fields

ship_to_new_address(*address_fields*)
Use a manually entered address as the shipping address

ship_to_user_address(*address*)
Use an user address (from an address book) as the shipping address.

shipping_method_code(*basket*)
Return the shipping method code

shipping_user_address_id()
Return user address id

use_free_shipping()
Set "free shipping" code to session

use_shipping_method(*code*)
Set shipping method code to session

user_address_id()
Return user address id

Customer

The customer app bundles communication with customers. This includes models to record product alerts and sent emails. It also contains the views that allow a customer to manage their data (profile information, shipping addresses, etc.)

Abstract models

class oscar.apps.customer.abstract_models.**AbstractCommunicationEventType**(**args*, ***kwargs*)
A 'type' of communication. Like an order confirmation email.

code
Code used for looking up this event programmatically.

get_messages(*ctx=None*)
Return a dict of templates with the context merged in

We look first at the field templates but fail over to a set of file templates that follow a conventional path.

name
Name is the friendly description of an event for use in the admin

class oscar.apps.customer.abstract_models.**AbstractEmail**(**args*, ***kwargs*)
This is a record of all emails sent to a customer. Normally, we only record order-related emails.

class oscar.apps.customer.abstract_models.**AbstractProductAlert**(**args*, ***kwargs*)
An alert for when a product comes back in stock

get_random_key()
Get a random generated key based on SHA-1 and email address

class oscar.apps.customer.abstract_models.**AbstractUser**(**args*, ***kwargs*)
An abstract base user suitable for use in Oscar projects.

This is basically a copy of the core AbstractUser model but without a username field

Forms

class oscar.apps.customer.forms.**ConfirmPasswordForm**(*user*, **args*, ***kwargs*)
Extends the standard django AuthenticationForm, to support 75 character usernames. 75 character usernames are needed to support the EmailOrUsername auth backend.

class oscar.apps.customer.forms.**EmailAuthenticationForm**(*host*, **args*, ***kwargs*)
Extends the standard django AuthenticationForm, to support 75 character usernames. 75 character usernames are needed to support the EmailOrUsername auth backend.

class oscar.apps.customer.forms.**PasswordResetForm**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*)
This form takes the same structure as its parent from django.contrib.auth

save(*domain_override=None*, *use_https=False*, *request=None*, ***kwargs*)
Generates a one-use only link for resetting password and sends to the user.

Views

class oscar.apps.customer.views.**AccountAuthView**(***kwargs*)
This is actually a slightly odd double form view that allows a customer to either login or register.

login_form_class
alias of EmailAuthenticationForm

class oscar.apps.customer.views.**AccountSummaryView**(***kwargs*)
View that exists for legacy reasons and customisability. It commonly gets called when the user clicks on "Account" in the navbar.

Oscar defaults to just redirecting to the profile summary page (and that redirect can be configured via OSCAR_ACCOUNT_REDIRECT_URL), but it's also likely you want to display an 'account overview' page or such like. The presence of this view allows just that, without having to change a lot of templates.

class oscar.apps.customer.views.**AddressChangeStatusView**(***kwargs*)
Sets an address as default_for_(billing|shipping)

class oscar.apps.customer.views.**AddressListView**(***kwargs*)
Customer address book

get_queryset()
Return customer's addresses

class oscar.apps.customer.views.**EmailDetailView**(***kwargs*)
Customer email

get_page_title()
Append email subject to page title

class oscar.apps.customer.views.**OrderHistoryView**(***kwargs*)
Customer order history

model
alias of Order

class oscar.apps.customer.views.**OrderLineView**(***kwargs*)
Customer order line

Alerts

The alerts module provides functionality that allows customers to sign up for email alerts when out-of-stock products come back in stock. A form for signing up is displayed on product detail pages when a product is not in stock.

If the `OSCAR_EAGER_ALERTS` setting is `True`, then alerts are sent as soon as affected stock records are updated. Alternatively, the management command `oscar_send_alerts` can be used to send alerts periodically.

The context for the alert email body contains a `hurry` variable that is set to `True` if the number of active alerts for a product is greater than the quantity of the product available in stock.

Alerts are sent using the Communication Event framework.

Dashboard

The dashboard is the backend interface for managing the store. That includes the product catalogue, orders and stock, offers etc. It is intended as a complete replacement of the Django admin interface. The app itself only contains a view that serves as a kind of homepage, and some logic for managing the navigation (in `nav.py`). There's several sub-apps that are responsible for managing the different parts of the Oscar store.

Permission-based dashboard

Staff users (users with `is_staff==True`) get access to all views in the dashboard. To better support Oscar's use for marketplace scenarios, the permission-based dashboard has been introduced. If a non-staff user has the `partner.dashboard_access` permission set, they are given access to a subset of views, and their access to products and orders is limited.

`AbstractPartner` instances have a *`users`* field. Prior to Oscar 0.6, this field was not used. Since Oscar 0.6, it is used solely for modelling dashboard access.

If a non-staff user with the `partner.dashboard_access` permission is in *`users`*, they can:

- Create products. It is enforced that at least one stock record's partner has the current user in `users`.
- Update products. At least one stock record must have the user in the stock record's partner's `users`.
- Delete and list products. Limited to products the user is allowed to update.
- Managing orders. Similar to products, a user get access if one of an order's lines is associated with a matching partner.

For many marketplace scenarios, it will make sense to ensure at checkout that a basket only contains lines from one partner. Please note that the dashboard currently ignores any other permissions, including Django's default permissions.

Note: The permission-based dashboard currently does not support parent or child products. Supporting this requires a modelling change. If you require this, please get in touch so we can first learn about your use case.

Abstract models

None.

Views

class oscar.apps.dashboard.views.**IndexView**(***kwargs*)

An overview view which displays several reports about the shop.

Supports the permission-based dashboard. It is recommended to add a index_nonstaff.html template because Oscar's default template will display potentially sensitive store information.

get_active_site_offers()

Return active conditional offers of type "site offer". The returned Queryset of site offers is filtered by end date greater then the current date.

get_active_vouchers()

Get all active vouchers. The returned Queryset of vouchers is filtered by end date greater then the current date.

get_hourly_report(*hours=24*, *segments=10*)

Get report of order revenue split up in hourly chunks. A report is generated for the last *hours* (default=24) from the current time. The report provides max_revenue of the hourly order revenue sum, y-range as the labeling for the y-axis in a template and order_total_hourly, a list of properties for hourly chunks. *segments* defines the number of labeling segments used for the y-axis when generating the y-axis labels (default=10).

get_number_of_promotions(*abstract_base=<class 'oscar.apps.promotions.models.AbstractPromotion'>*)

Get the number of promotions for all promotions derived from *abstract_base*. All subclasses of *abstract_base* are queried and if another abstract base class is found this method is executed recursively.

get_open_baskets(*filters=None*)

Get all open baskets. If *filters* dictionary is provided they will be applied on all open baskets and return only filtered results.

Offers

Oscar ships with a powerful and flexible offers engine which is contained in the offers app. 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 an 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 maximum 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 (the value is implicitly 1)

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...

Abstract models

class `oscar.apps.offer.abstract_models.`**`AbstractCondition`**(**args*, ***kwargs*)

A condition for an offer to be applied. You can either specify a custom proxy class, or need to specify a type, range and value.

`can_apply_condition`(*line*)

Determines whether the condition can be applied to a given basket line

`description`

A description of the condition. Defaults to the name. May contain HTML.

`get_applicable_lines`(*offer*, *basket*, *most_expensive_first=True*)

Return line data for the lines that can be consumed by this condition

`is_partially_satisfied`(*offer*, *basket*)

Determine if the basket partially meets the condition. This is useful for up-selling messages to entice customers to buy something more in order to qualify for an offer.

`is_satisfied`(*offer*, *basket*)

Determines whether a given basket meets this condition. This is stubbed in this top-class object. The subclassing proxies are responsible for implementing it correctly.

name
A plaintext description of the condition. Every proxy class has to implement it.

This is used in the dropdowns within the offer dashboard.

proxy()
Return the proxy model

class oscar.apps.offer.abstract_models.**AbstractConditionalOffer**(**args*, ***kwargs*)
A conditional offer (eg buy 1, get 10% off)

apply_benefit(*basket*)
Applies the benefit to the given basket and returns the discount.

apply_deferred_benefit(*basket*, *order*, *application*)
Applies any deferred benefits. These are things like adding loyalty points to someone's account.

availability_description()
Return a description of when this offer is available

get_max_applications(*user=None*)
Return the number of times this offer can be applied to a basket for a given user.

is_available(*user=None*, *test_date=None*)
Test whether this offer is available to be used

products()
Return a queryset of products in this offer

class oscar.apps.offer.abstract_models.**AbstractRange**(**args*, ***kwargs*)
Represents a range of products that can be used within an offer.

Ranges only support adding parent or stand-alone products. Offers will consider child products automatically.

add_product(*product*, *display_order=None*)
Add product to the range

When adding product that is already in the range, prevent re-adding it. If display_order is specified, update it.

Default display_order for a new product in the range is 0; this puts the product at the top of the list.

all_products()
Return a queryset containing all the products in the range

This includes included_products plus the products contained in the included classes and categories, minus the products in excluded_products.

contains(*product*)
Check whether the passed product is part of this range.

contains_product(*product*)
Check whether the passed product is part of this range.

is_editable
Test whether this range can be edited in the dashboard.

is_reorderable
Test whether products for the range can be re-ordered.

remove_product(*product*)
Remove product from range. To save on queries, this function does not check if the product is in fact in the range.

class oscar.apps.offer.abstract_models.**AbstractRangeProduct**(**args*, ***kwargs*)
Allow ordering products inside ranges Exists to allow customising.

Models

class oscar.apps.offer.models.**BasketDiscount**(*amount*)
For when an offer application leads to a simple discount off the basket's total

class oscar.apps.offer.models.**ShippingDiscount**
For when an offer application leads to a discount from the shipping cost

class oscar.apps.offer.models.**PostOrderAction**(*description*)
For when an offer condition is met but the benefit is deferred until after the order has been placed. Eg buy 2 books and get 100 loyalty points.

class oscar.apps.offer.models.**ConditionalOffer**(*id*, *name*, *slug*, *description*, *offer_type*, *status*, *condition*, *benefit*, *priority*, *start_datetime*, *end_datetime*, *max_global_applications*, *max_user_applications*, *max_basket_applications*, *max_discount*, *total_discount*, *num_applications*, *num_orders*, *redirect_url*, *date_created*)

class oscar.apps.offer.models.**Benefit**(*id*, *range*, *type*, *value*, *max_affected_items*, *proxy_class*)

class oscar.apps.offer.models.**Condition**(*id*, *range*, *type*, *value*, *proxy_class*)

class oscar.apps.offer.models.**Range**(*id*, *name*, *slug*, *description*, *is_public*, *includes_all_products*, *proxy_class*, *date_created*)

class oscar.apps.offer.models.**RangeProduct**(*id*, *range*, *product*, *display_order*)

class oscar.apps.offer.models.**RangeProductFileUpload**(*id*, *range*, *filepath*, *size*, *uploaded_by*, *date_uploaded*, *status*, *error_message*, *date_processed*, *num_new_skus*, *num_unknown_skus*, *num_duplicate_skus*)

class oscar.apps.offer.models.**PercentageDiscountBenefit**(**args*, ***kwargs*)
An offer benefit that gives a percentage discount

class oscar.apps.offer.models.**AbsoluteDiscountBenefit**(**args*, ***kwargs*)
An offer benefit that gives an absolute discount

class oscar.apps.offer.models.**FixedPriceBenefit**(**args*, ***kwargs*)
An offer benefit that gives the items in the condition for a fixed price. This is useful for "bundle" offers.

Note that we ignore the benefit range here and only give a fixed price for the products in the condition range. The condition cannot be a value condition.

We also ignore the max_affected_items setting.

class oscar.apps.offer.models.**ShippingBenefit**(*id*, *range*, *type*, *value*, *max_affected_items*, *proxy_class*)

class oscar.apps.offer.models.**MultibuyDiscountBenefit**(*id*, *range*, *type*, *value*, *max_affected_items*, *proxy_class*)

class oscar.apps.offer.models.**ShippingAbsoluteDiscountBenefit**(*id*, *range*, *type*, *value*, *max_affected_items*, *proxy_class*)

class oscar.apps.offer.models.**ShippingFixedPriceBenefit**(*id*, *range*, *type*, *value*, *max_affected_items*, *proxy_class*)

class oscar.apps.offer.models.**ShippingPercentageDiscountBenefit**(*id*, *range*, *type*, *value*, *max_affected_items*, *proxy_class*)

class oscar.apps.offer.models.**CountCondition**(**args*, ***kwargs*)
An offer condition dependent on the NUMBER of matching items from the basket.

consume_items(*offer*, *basket*, *affected_lines*)
Marks items within the basket lines as consumed so they can't be reused in other offers.

Basket The basket

Affected_lines The lines that have been affected by the discount. This should be list of tuples (line, discount, qty)

is_satisfied(*offer*, *basket*)
Determines whether a given basket meets this condition

class oscar.apps.offer.models.**CoverageCondition**(**args*, ***kwargs*)
An offer condition dependent on the number of DISTINCT matching items from the basket.

consume_items(*offer*, *basket*, *affected_lines*)
Marks items within the basket lines as consumed so they can't be reused in other offers.

is_satisfied(*offer*, *basket*)
Determines whether a given basket meets this condition

class oscar.apps.offer.models.**ValueCondition**(**args*, ***kwargs*)
An offer condition dependent on the VALUE of matching items from the basket.

consume_items(*offer*, *basket*, *affected_lines*)
Marks items within the basket lines as consumed so they can't be reused in other offers.

We allow lines to be passed in as sometimes we want them sorted in a specific order.

is_satisfied(*offer*, *basket*)
Determine whether a given basket meets this condition

Views

Order

The order app handles processing of orders.

Abstract models

class oscar.apps.order.abstract_models.**AbstractCommunicationEvent**(**args*, ***kwargs*)
An order-level event involving a communication to the customer, such as an confirmation email being sent.

class oscar.apps.order.abstract_models.**AbstractLine**(**args*, ***kwargs*)
An order line

classmethod all_statuses()
Return all possible statuses for an order line

available_statuses()
Return all possible statuses that this order line can move to

description
Returns a description of this line including details of any line attributes.

get_event_quantity(*event*)
Fetches the ShippingEventQuantity instance for this line

Exists as a separate method so it can be overridden to avoid the DB query that's caused by get().

has_shipping_event_occurred(*event_type*, *quantity=None*)
Test whether this line has passed a given shipping event

is_available_to_reorder(*basket*, *strategy*)
Test if this line can be re-ordered using the passed strategy and basket

is_payment_event_permitted(*event_type*, *quantity*)
Test whether a payment event with the given quantity is permitted.

Allow each payment event type to occur only once per quantity.

is_shipping_event_permitted(*event_type*, *quantity*)
Test whether a shipping event with the given quantity is permitted

This method should normally be overriden to ensure that the prerequisite shipping events have been passed for this line.

payment_event_quantity(*event_type*)
Return the quantity of this line that has been involved in a payment event of the passed type.

pipeline = {}
Order status pipeline. This should be a dict where each (key, value) corresponds to a status and the possible statuses that can follow that one.

set_status(*new_status*)
Set a new status for this line

If the requested status is not valid, then `InvalidLineStatus` is raised.

shipping_event_breakdown
Returns a dict of shipping events that this line has been through

shipping_event_quantity(*event_type*)
Return the quantity of this line that has been involved in a shipping event of the passed type.

shipping_status
Returns a string summary of the shipping status of this line

class oscar.apps.order.abstract_models.**AbstractLineAttribute**(**args*, ***kwargs*)
An attribute of a line

class oscar.apps.order.abstract_models.**AbstractLinePrice**(**args*, ***kwargs*)
For tracking the prices paid for each unit within a line.

This is necessary as offers can lead to units within a line having different prices. For example, one product may be sold at 50% off as it's part of an offer while the remainder are full price.

class oscar.apps.order.abstract_models.**AbstractOrder**(**args*, ***kwargs*)
The main order model

classmethod all_statuses()
Return all possible statuses for an order

available_statuses()
Return all possible statuses that this order can move to

basket_total_before_discounts_excl_tax
Return basket total excluding tax but before discounts are applied

basket_total_before_discounts_incl_tax
Return basket total including tax but before discounts are applied

basket_total_excl_tax
Return basket total excluding tax

basket_total_incl_tax
Return basket total including tax

cascade = {'Cancelled': 'Cancelled', 'Complete': 'Shipped', 'Being processed': 'Being processed'}
Order status cascade pipeline. This should be a dict where each (key, value) pair corresponds to an *order* status and the corresponding *line* status that needs to be set when the order is set to the new status

check_deprecated_verification_hash(*hash_to_check*)
Backward compatible check for md5 hashes that were generated in Oscar 1.5 and lower.

This must explicitly be enabled by setting OSCAR_DEPRECATED_ORDER_VERIFY_KEY, which must not be equal to SECRET_KEY - i.e., the project must have changed its SECRET_KEY since this change was applied.

TODO: deprecate this method in Oscar 2.0, and remove it in Oscar 2.1.

check_verification_hash(*hash_to_check*)
Checks the received verification hash against this order number. Returns False if the verification failed, True otherwise.

num_items
Returns the number of items in this order.

pipeline = {'Cancelled': (), 'Being processed': ('Complete', 'Cancelled'), 'Pending': ('Being processed', 'Cancelled'), '
Order status pipeline. This should be a dict where each (key, value) # – corresponds to a status and a list of possible statuses that can follow that one.

set_status(*new_status*)
Set a new status for this order.

If the requested status is not valid, then `InvalidOrderStatus` is raised.

shipping_status
Return the last complete shipping event for this order.

total_discount_incl_tax
The amount of discount this order received

class oscar.apps.order.abstract_models.**AbstractOrderDiscount**(**args*, ***kwargs*)
A discount against an order.

Normally only used for display purposes so an order can be listed with discounts displayed separately even though in reality, the discounts are applied at the line level.

This has evolved to be a slightly misleading class name as this really track benefit applications which aren't necessarily discounts.

class oscar.apps.order.abstract_models.**AbstractOrderNote**(**args*, ***kwargs*)
A note against an order.

This are often used for audit purposes too. IE, whenever an admin makes a change to an order, we create a note to record what happened.

class oscar.apps.order.abstract_models.**AbstractPaymentEvent**(**args*, ***kwargs*)
A payment event for an order

For example:

•All lines have been paid for

•2 lines have been refunded

class oscar.apps.order.abstract_models.**AbstractPaymentEventType**(**args*, ***kwargs*)
Payment event types are things like 'Paid', 'Failed', 'Refunded'.

These are effectively the transaction types.

class oscar.apps.order.abstract_models.**AbstractShippingEvent**(**args*, ***kwargs*)
An event is something which happens to a group of lines such as 1 item being dispatched.

class oscar.apps.order.abstract_models.**AbstractShippingEventType**(**args*, ***kwargs*)
A type of shipping/fulfillment event

Eg: 'Shipped', 'Cancelled', 'Returned'

class oscar.apps.order.abstract_models.**PaymentEventQuantity**(**args*, ***kwargs*)
A "through" model linking lines to payment events

class oscar.apps.order.abstract_models.**ShippingEventQuantity**(**args*, ***kwargs*)
A "through" model linking lines to shipping events.

This exists to track the quantity of a line that is involved in a particular shipping event.

Order processing

class oscar.apps.order.processing.**EventHandler**(*user=None*)
Handle requested order events.

This is an important class: it houses the core logic of your shop's order processing pipeline.

are_stock_allocations_available(*lines*, *line_quantities*)
Check whether stock records still have enough stock to honour the requested allocations.

calculate_payment_event_subtotal(*event_type*, *lines*, *line_quantities*)
Calculate the total charge for the passed event type, lines and line quantities.

This takes into account the previous prices that have been charged for this event.

Note that shipping is not including in this subtotal. You need to subclass and extend this method if you want to include shipping costs.

cancel_stock_allocations(*order*, *lines*, *line_quantities*)
Cancel the stock allocations for the passed lines

consume_stock_allocations(*order*, *lines*, *line_quantities*)
Consume the stock allocations for the passed lines

handle_order_status_change(*order*, *new_status*, *note_msg=None*)
Handle a requested order status change

This method is not normally called directly by client code. The main use-case is when an order is cancelled, which in some ways could be viewed as a shipping event affecting all lines.

handle_payment_event(*order*, *event_type*, *amount*, *lines=None*, *line_quantities=None*, ***kwargs*)
Handle a payment event for a given order.

These should normally be called as part of handling a shipping event. It is rare to call to this method directly. It does make sense for refunds though where the payment event may be unrelated to a particular shipping event and doesn't directly correspond to a set of lines.

handle_shipping_event(*order*, *event_type*, *lines*, *line_quantities*, ***kwargs*)
Handle a shipping event for a given order.

This is most common entry point to this class - most of your order processing should be modelled around shipping events. Shipping events can be used to trigger payment and communication events.

You will generally want to override this method to implement the specifics of you order processing pipeline.

have_lines_passed_shipping_event(*order*, *lines*, *line_quantities*, *event_type*)
Test whether the passed lines and quantities have been through the specified shipping event.

This is useful for validating if certain shipping events are allowed (ie you can't return something before it has shipped).

validate_shipping_event(*order*, *event_type*, *lines*, *line_quantities*, ***kwargs*)
Test if the requested shipping event is permitted.

If not, raise InvalidShippingEvent

Utils

class oscar.apps.order.utils.**OrderCreator**
Places the order by writing out the various models

create_additional_line_models(*order*, *order_line*, *basket_line*)
Empty method designed to be overridden.

Some applications require additional information about lines, this method provides a clean place to create additional models that relate to a given line.

create_discount_model(*order*, *discount*)
Create an order discount model for each offer application attached to the basket.

create_line_attributes(*order*, *order_line*, *basket_line*)
Creates the batch line attributes.

create_line_models(*order*, *basket_line*, *extra_line_fields=None*)
Create the batch line model.

You can set extra fields by passing a dictionary as the extra_line_fields value

create_line_price_models(*order*, *order_line*, *basket_line*)
Creates the batch line price models

create_order_model(*user*, *basket*, *shipping_address*, *shipping_method*, *shipping_charge*, *billing_address*, *total*, *order_number*, *status*, *request=None*, ***extra_order_fields*)
Create an order model.

place_order(*basket*, *total*, *shipping_method*, *shipping_charge*, *user=None*, *shipping_address=None*, *billing_address=None*, *order_number=None*, *status=None*, *request=None*, ***kwargs*)
Placing an order involves creating all the relevant models based on the basket and session data.

record_voucher_usage(*order*, *voucher*, *user*)
Updates the models that care about this voucher.

update_stock_records(*line*)
Update any relevant stock records for this order line

class oscar.apps.order.utils.**OrderNumberGenerator**
Simple object for generating order numbers.

We need this as the order number is often required for payment which takes place before the order model has been created.

order_number(*basket*)
Return an order number for a given basket

Partner

The partner app mostly provides three abstract models. *oscar.apps.partner.abstract_models.AbstractPartner* and *oscar.apps.partner.abstract_models.AbstractStockRecord* are essential parts of Oscar's catalogue management.

Abstract models

class oscar.apps.partner.abstract_models.**AbstractPartner**(**args*, ***kwargs*)
A fulfillment partner. An individual or company who can fulfil products. E.g. for physical goods, somebody with a warehouse and means of delivery.

Creating one or more instances of the Partner model is a required step in setting up an Oscar deployment. Many Oscar deployments will only have one fulfillment partner.

get_address_for_stockrecord(*stockrecord*)
Stock might be coming from different warehouses. Overriding this function allows selecting the correct PartnerAddress for the record. That can be useful when determining tax.

primary_address
Returns a partners primary address. Usually that will be the headquarters or similar.

This is a rudimentary implementation that raises an error if there's more than one address. If you actually want to support multiple addresses, you will likely need to extend PartnerAddress to have some field or flag to base your decision on.

users
A partner can have users assigned to it. This is used for access modelling in the permission-based dashboard

class oscar.apps.partner.abstract_models.**AbstractStockAlert**(**args*, ***kwargs*)
A stock alert. E.g. used to notify users when a product is 'back in stock'.

class oscar.apps.partner.abstract_models.**AbstractStockRecord**(**args*, ***kwargs*)
A stock record.

This records information about a product from a fulfilment partner, such as their SKU, the number they have in stock and price information.

Stockrecords are used by 'strategies' to determine availability and pricing information for the customer.

allocate(*quantity*)
Record a stock allocation.

This normally happens when a product is bought at checkout. When the product is actually shipped, then we 'consume' the allocation.

consume_allocation(*quantity*)
Consume a previous allocation

This is used when an item is shipped. We remove the original allocation and adjust the number in stock accordingly

cost_price
Cost price is the price charged by the fulfilment partner. It is not used (by default) in any price calculations but is often used in reporting so merchants can report on their profit margin.

is_allocation_consumption_possible(*quantity*)
Test if a proposed stock consumption is permitted

low_stock_threshold
Threshold for low-stock alerts. When stock goes beneath this threshold, an alert is triggered so warehouse managers can order more.

net_stock_level
The effective number in stock (eg available to buy).

This is correct property to show the customer, not the num_in_stock field as that doesn't account for allocations. This can be negative in some unusual circumstances

num_allocated
The amount of stock allocated to orders but not fed back to the master stock system. A typical stock update process will set the num_in_stock variable to a new value and reset num_allocated to zero

num_in_stock
Number of items in stock

partner_sku
The fulfilment partner will often have their own SKU for a product, which we store here. This will sometimes be the same the product's UPC but not always. It should be unique per partner. See also http://en.wikipedia.org/wiki/Stock-keeping_unit

price_retail
Retail price for this item. This is simply the recommended price from the manufacturer. If this is used, it is for display purposes only. This prices is the NOT the price charged to the customer.

Strategy classes

class `oscar.apps.partner.strategy.`**Base**(*request=None*)
The base strategy class

Given a product, strategies are responsible for returning a `PurchaseInfo` instance which contains:

- The appropriate stockrecord for this customer
- A pricing policy instance
- An availability policy instance

fetch_for_line(*line*, *stockrecord=None*)
Given a basket line instance, fetch a `PurchaseInfo` instance.

This method is provided to allow purchase info to be determined using a basket line's attributes. For instance, "bundle" products often use basket line attributes to store SKUs of contained products. For such products, we need to look at the availability of each contained product to determine overall availability.

fetch_for_parent(*product*)
Given a parent product, fetch a `StockInfo` instance

fetch_for_product(*product*, *stockrecord=None*)
Given a product, return a `PurchaseInfo` instance.

The `PurchaseInfo` class is a named tuple with attributes:

- `price`: a pricing policy object.
- `availability`: an availability policy object.
- `stockrecord`: the stockrecord that is being used

If a stockrecord is passed, return the appropriate `PurchaseInfo` instance for that product and stockrecord is returned.

class `oscar.apps.partner.strategy.`**Default**(*request=None*)
Default stock/price strategy that uses the first found stockrecord for a product, ensures that stock is available (unless the product class indicates that we don't need to track stock) and charges zero tax.

class `oscar.apps.partner.strategy.`**DeferredTax**
Pricing policy mixin for use with the `Structured` base strategy. This mixin does not specify the product tax and is suitable to territories where tax isn't known until late in the checkout process.

class `oscar.apps.partner.strategy.`**FixedRateTax**
Pricing policy mixin for use with the `Structured` base strategy. This mixin applies a fixed rate tax to the base price from the product's stockrecord. The price_incl_tax is quantized to two decimal places. Rounding behaviour is Decimal's default

get_exponent(*stockrecord*)
This method serves as hook to be able to plug in support for a varying exponent based on the currency.

TODO: Needs tests.

get_rate(*product*, *stockrecord*)
This method serves as hook to be able to plug in support for varying tax rates based on the product.

TODO: Needs tests.

class `oscar.apps.partner.strategy.`**NoTax**
Pricing policy mixin for use with the `Structured` base strategy. This mixin specifies zero tax and uses the `price_excl_tax` from the stockrecord.

class `oscar.apps.partner.strategy.`**PurchaseInfo**(*price*, *availability*, *stockrecord*)

availability
Alias for field number 1

price
Alias for field number 0

stockrecord
Alias for field number 2

class `oscar.apps.partner.strategy.`**Selector**
Responsible for returning the appropriate strategy class for a given user/session.

This can be called in three ways:

1. Passing a request and user. This is for determining prices/availability for a normal user browsing the site.
2. Passing just the user. This is for offline processes that don't have a request instance but do know which user to determine prices for.
3. Passing nothing. This is for offline processes that don't correspond to a specific user. Eg, determining a price to store in a search index.

strategy(*request=None*, *user=None*, ***kwargs*)
Return an instanticated strategy instance

class oscar.apps.partner.strategy.**StockRequired**
Availability policy mixin for use with the `Structured` base strategy. This mixin ensures that a product can only be bought if it has stock available (if stock is being tracked).

class oscar.apps.partner.strategy.**Structured**(*request=None*)
A strategy class which provides separate, overridable methods for determining the 3 things that a `PurchaseInfo` instance requires:

1. A stockrecord
2. A pricing policy
3. An availability policy

availability_policy(*product*, *stockrecord*)
Return the appropriate availability policy

fetch_for_product(*product*, *stockrecord=None*)
Return the appropriate `PurchaseInfo` instance.

This method is not intended to be overridden.

pricing_policy(*product*, *stockrecord*)
Return the appropriate pricing policy

select_children_stockrecords(*product*)
Select appropriate stock record for all children of a product

select_stockrecord(*product*)
Select the appropriate stockrecord

class oscar.apps.partner.strategy.**UK**(*request=None*)
Sample strategy for the UK that:

- **uses the first stockrecord for each product (effectively assuming** there is only one).
- requires that a product has stock available to be bought
- applies a fixed rate of tax on all products

This is just a sample strategy used for internal development. It is not recommended to be used in production, especially as the tax rate is hard-coded.

class oscar.apps.partner.strategy.**US**(*request=None*)
Sample strategy for the US.

- uses the first stockrecord for each product (effectively assuming there is only one).
- requires that a product has stock available to be bought
- doesn't apply a tax to product prices (normally this will be done after the shipping address is entered).

This is just a sample one used for internal development. It is not recommended to be used in production.

class oscar.apps.partner.strategy.**UseFirstStockRecord**
Stockrecord selection mixin for use with the `Structured` base strategy. This mixin picks the first (normally only) stockrecord to fulfil a product.

This is backwards compatible with Oscar<0.6 where only one stockrecord per product was permitted.

Pricing policies

class oscar.apps.partner.prices.**Base**
The interface that any pricing policy must support

currency = None
Price currency (3 char code)

excl_tax = None
Price excluding tax

exists = False
Whether any prices exist

incl_tax = None
Price including tax

is_tax_known = False
Whether tax is known

retail = None
Retail price

tax = None
Price tax

class oscar.apps.partner.prices.**FixedPrice**(*currency*, *excl_tax*, *tax=None*)
This should be used for when the price of a product is known in advance.

It can work for when tax isn't known (like in the US).

Note that this price class uses the tax-exclusive price for offers, even if the tax is known. This may not be what you want. Use the TaxInclusiveFixedPrice class if you want offers to use tax-inclusive prices.

class oscar.apps.partner.prices.**TaxInclusiveFixedPrice**(*currency*, *excl_tax*, *tax*)
Specialised version of FixedPrice that must have tax passed. It also specifies that offers should use the tax-inclusive price (which is the norm in the UK).

class oscar.apps.partner.prices.**Unavailable**
This should be used as a pricing policy when a product is unavailable and no prices are known.

Availability policies

class oscar.apps.partner.availability.**Available**
For when a product is always available, irrespective of stock level.

This might be appropriate for digital products where stock doesn't need to be tracked and the product is always available to buy.

class oscar.apps.partner.availability.**Base**
Base availability policy.

code = ''
Availability code. This is used for HTML classes

dispatch_date = None
When this item should be dispatched

is_available_to_buy
Test if this product is available to be bought. This is used for validation when a product is added to a user's basket.

is_purchase_permitted(*quantity*)
Test whether a proposed purchase is allowed

Should return a boolean and a reason

message = ''
A description of the availability of a product. This is shown on the product detail page. Eg "In stock", "Out of stock" etc

short_message
A shorter version of the availability message, suitable for showing on browsing pages.

class oscar.apps.partner.availability.**StockRequired**(*num_available*)
Allow a product to be bought while there is stock. This policy is instantiated with a stock number (num_available). It ensures that the product is only available to buy while there is stock available.

This is suitable for physical products where back orders (eg allowing purchases when there isn't stock available) are not permitted.

class oscar.apps.partner.availability.**Unavailable**
Policy for when a product is unavailable

Payment

The payment app contains models that capture how orders are paid for. It does not have any views.

Abstract models

class oscar.apps.payment.abstract_models.**AbstractBankcard**(**args*, ***kwargs*)
Model representing a user's bankcard. This is used for two purposes:

1. The bankcard form will return an instance of this model that can be used with payment gateways. In this scenario, the instance will have additional attributes (start_date, issue_number, ccv) that payment gateways need but that we don't save.
2. To keep a record of a user's bankcards and allow them to be re-used. This is normally done using the 'partner reference'.

> **Warning:** Some of the fields of this model (name, expiry_date) are considered "cardholder data" under PCI DSS v2. Hence, if you use this model and store those fields then the requirements for PCI compliance will be more stringent.

class oscar.apps.payment.abstract_models.**AbstractSource**(**args*, ***kwargs*)
A source of payment for an order.

This is normally a credit card which has been pre-authed for the order amount, but some applications will allow orders to be paid for using multiple sources such as cheque, credit accounts, gift cards. Each payment source will have its own entry.

This source object tracks how much money has been authorised, debited and refunded, which is useful when payment takes place in multiple stages.

allocate(*amount*, *reference=''*, *status=''*)
Convenience method for ring-fencing money against this source

amount_available_for_refund
Return the amount available to be refunded

balance
Return the balance of this source

create_deferred_transaction(*txn_type*, *amount*, *reference=None*, *status=None*)
Register the data for a transaction that can't be created yet due to FK constraints. This happens at checkout where create an payment source and a transaction but can't save them until the order model exists.

debit(*amount=None*, *reference=''*, *status=''*)
Convenience method for recording debits against this source

refund(*amount*, *reference=''*, *status=''*)
Convenience method for recording refunds against this source

class oscar.apps.payment.abstract_models.**AbstractSourceType**(**args*, ***kwargs*)
A type of payment source.

This could be an external partner like PayPal or DataCash, or an internal source such as a managed account.

class oscar.apps.payment.abstract_models.**AbstractTransaction**(**args*, ***kwargs*)
A transaction for a particular payment source.

These are similar to the payment events within the order app but model a slightly different aspect of payment. Crucially, payment sources and transactions have nothing to do with the lines of the order while payment events do.

For example: * A 'pre-auth' with a bankcard gateway * A 'settle' with a credit provider (see django-oscar-accounts)

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.

Models

class oscar.apps.promotions.models.**AbstractProductList**(**args*, ***kwargs*)
Abstract superclass for promotions which are essentially a list of products.

class oscar.apps.promotions.models.**AbstractPromotion**(**args*, ***kwargs*)
Abstract base promotion that defines the interface that subclasses must implement.

template_name()
Returns the template to use to render this promotion.

class oscar.apps.promotions.models.**AutomaticProductList**(*id*, *name*, *description*, *link_url*, *link_text*, *date_created*, *method*, *num_products*)

class oscar.apps.promotions.models.**HandPickedProductList**(**args*, ***kwargs*)
A hand-picked product list is a list of manually selected products.

class oscar.apps.promotions.models.**Image**(**args*, ***kwargs*)
An image promotion is simply a named image which has an optional link to another part of the site (or another site).

This can be used to model both banners and pods.

class oscar.apps.promotions.models.**KeywordPromotion**(**args*, ***kwargs*)
A promotion linked to a specific keyword.

This can be used on a search results page to show promotions linked to a particular keyword.

class oscar.apps.promotions.models.**MultiImage**(**args*, ***kwargs*)
A multi-image promotion is simply a collection of image promotions that are rendered in a specific way. This models things like rotating banners.

class oscar.apps.promotions.models.**OrderedProduct**(*id*, *list*, *product*, *display_order*)

class oscar.apps.promotions.models.**OrderedProductList**(*id*, *name*, *description*, *link_url*, *link_text*, *date_created*, *handpickedproductlist_ptr*, *tabbed_block*, *display_order*)

class oscar.apps.promotions.models.**PagePromotion**(**args*, ***kwargs*)
A promotion embedded on a particular page.

class oscar.apps.promotions.models.**RawHTML**(**args*, ***kwargs*)
Simple promotion - just raw HTML

class oscar.apps.promotions.models.**SingleProduct**(*id*, *name*, *product*, *description*, *date_created*)

class oscar.apps.promotions.models.**TabbedBlock**(*id*, *name*, *date_created*)

Views

class oscar.apps.promotions.views.**HomeView**(***kwargs*)
This is the home page and will typically live at /

class oscar.apps.promotions.views.**RecordClickView**(***kwargs*)
Simple RedirectView that helps recording clicks made on promotions

Search

Oscar provides a search view that extends Haystack's `FacetedSearchView` to provide better support for faceting.

- Facets are configured using the `OSCAR_SEARCH_FACETS` setting, which is used to configure the `SearchQuerySet` instance within the search application class.
- A simple search form is injected into each template context using a context processor `oscar.apps.search.context_processors.search_form`.

Views

class oscar.apps.search.views.**FacetedSearchView**(**args*, ***kwargs*)
A modified version of Haystack's FacetedSearchView

Note that facets are configured when the SearchQuerySet is initialised. This takes place in the search application class.

See https://django-haystack.readthedocs.io/en/v2.1.0/views_and_forms.html#facetedsearchform # noqa

Forms

class oscar.apps.search.forms.**BrowseCategoryForm**(**args*, ***kwargs*)
Variant of SearchForm that returns all products (instead of none) if no query is specified.

class oscar.apps.search.forms.**SearchForm**(**args*, ***kwargs*)
In Haystack, the search form is used for interpreting and sub-filtering the SQS.

selected_multi_facets
Validate and return the selected facets

class oscar.apps.search.forms.**SearchInput**(*attrs=None*)
Defining a search type widget

This is an HTML5 thing and works nicely with Safari, other browsers default back to using the default "text" type

Utils

oscar.apps.search.facets.**base_sqs**()
Return the base SearchQuerySet for Haystack searches.

Shipping

See *How to configure shipping* for details on how shipping works in Oscar.

Methods

class oscar.apps.shipping.methods.**Base**
Shipping method interface class

This is the superclass to the classes in methods.py, and a de-facto superclass to the classes in models.py. This allows using all shipping methods interchangeably (aka polymorphism).

The interface is all properties.

calculate(*basket*)
Return the shipping charge for the given basket

code = '__default__'
Used to store this method in the session. Each shipping method should

description = ''
A more detailed description of the shipping method shown to the customer

discount(*basket*)
Return the discount on the standard shipping charge

is_discounted = False
Whether the charge includes a discount

name = 'Default shipping'
The name of the shipping method, shown to the customer during checkout

class oscar.apps.shipping.methods.**FixedPrice**(*charge_excl_tax=None*, *charge_incl_tax=None*)
This shipping method indicates that shipping costs a fixed price and requires no special calculation.

class oscar.apps.shipping.methods.**Free**
This shipping method specifies that shipping is free.

class oscar.apps.shipping.methods.**NoShippingRequired**
This is a special shipping method that indicates that no shipping is actually required (eg for digital goods).

class oscar.apps.shipping.methods.**OfferDiscount**(*method*, *offer*)
Wrapper class that applies a discount to an existing shipping method's charges.

class oscar.apps.shipping.methods.**TaxExclusiveOfferDiscount**(*method*, *offer*)
Wrapper class which extends OfferDiscount to be exclusive of tax.

class oscar.apps.shipping.methods.**TaxInclusiveOfferDiscount**(*method*, *offer*)
Wrapper class which extends OfferDiscount to be inclusive of tax.

calculate_excl_tax(*base_charge*, *incl_tax*)
Return the charge excluding tax (but including discount).

Models

class oscar.apps.shipping.models.**OrderAndItemCharges**(*id*, *code*, *name*, *description*, *price_per_order*, *price_per_item*, *free_shipping_threshold*)

class oscar.apps.shipping.models.**WeightBased**(*id*, *code*, *name*, *description*, *default_weight*)

class oscar.apps.shipping.models.**WeightBand**(*id*, *method*, *upper_limit*, *charge*)

Repository

class oscar.apps.shipping.repository.**Repository**
Repository class responsible for returning ShippingMethod objects for a given user, basket etc

apply_shipping_offer(*basket*, *method*, *offer*)
Wrap a shipping method with an offer discount wrapper (as long as the shipping charge is non-zero).

apply_shipping_offers(*basket*, *methods*)
Apply shipping offers to the passed set of methods

get_available_shipping_methods(*basket*, *shipping_addr=None*, ***kwargs*)
Return a list of all applicable shipping method instances for a given basket, address etc. This method is intended to be overridden.

get_default_shipping_method(*basket*, *shipping_addr=None*, ***kwargs*)
Return a 'default' shipping method to show on the basket page to give the customer an indication of what their order will cost.

get_shipping_methods(*basket*, *shipping_addr=None*, ***kwargs*)
Return a list of all applicable shipping method instances for a given basket, address etc.

Voucher

Oscar ships with broad support for vouchers, which are handled by this app.

Abstract models

class oscar.apps.voucher.abstract_models.**AbstractVoucher**(**args*, ***kwargs*)
A voucher. This is simply a link to a collection of offers.

Note that there are three possible "usage" modes: (a) Single use (b) Multi-use (c) Once per customer

Oscar enforces those modes by creating VoucherApplication instances when a voucher is used for an order.

benefit
Returns the first offer's benefit instance.

A voucher is commonly only linked to one offer. In that case, this helper can be used for convenience.

is_active(*test_datetime=None*)
Test whether this voucher is currently active.

is_available_to_user(*user=None*)
Test whether this voucher is available to the passed user.

Returns a tuple of a boolean for whether it is successful, and a availability message.

is_expired()
Test whether this voucher has passed its expiration date

record_discount(*discount*)
Record a discount that this offer has given

record_usage(*order*, *user*)
Records a usage of this voucher in an order.

class oscar.apps.voucher.abstract_models.**AbstractVoucherApplication**(**args*, ***kwargs*)
For tracking how often a voucher has been used in an order.

This is used to enforce the voucher usage mode in Voucher.is_available_to_user, and created in Voucher.record_usage.

Views

None.

Wishlists

The wishlists app allows signed-in users to create one or more wishlists. A user can add a product to their wishlist from the product detail page and manage their lists in the account section.

The wishlists app is wired up as a subapp of *Customer*.

Note: Please note that currently only private wishlists are supported. The hooks and fields for public (as in general public) and shared (as in access via an obfuscated link) are there, but the UI hasn't been designed yet.

Abstract models

class oscar.apps.wishlists.abstract_models.**AbstractLine**(**args*, ***kwargs*)
One entry in a wish list. Similar to order lines or basket lines.

title
Store the title in case product gets deleted

class oscar.apps.wishlists.abstract_models.**AbstractWishList**(**args*, ***kwargs*)
Represents a user's wish lists of products.

A user can have multiple wish lists, move products between them, etc.

add(*product*)
Add a product to this wishlist

key
This key acts as primary key and is used instead of an int to make it harder to guess

classmethod random_key(*length=6*)
Get a unique random generated key based on SHA-1 and owner

Views

class oscar.apps.customer.wishlists.views.**LineMixin**
Handles fetching both a wish list and a product Views using this mixin must be passed two keyword arguments:

- key: The key of a wish list
- line_pk: The primary key of the wish list line

or

- product_pk: The primary key of the product

class oscar.apps.customer.wishlists.views.**WishListAddProduct**(***kwargs*)
Adds a product to a wish list.

- If the user doesn't already have a wishlist then it will be created for them.
- If the product is already in the wish list, its quantity is increased.

class oscar.apps.customer.wishlists.views.**WishListCreateView**(***kwargs*)
Create a new wishlist

If a product ID is assed as a kwargs, then this product will be added to the wishlist.

model
alias of WishList

class oscar.apps.customer.wishlists.views.**WishListCreateWithProductView**(***kwargs*)
Create a wish list and immediately add a product to it

class oscar.apps.customer.wishlists.views.**WishListDetailView**(***kwargs*)
This view acts as a DetailView for a wish list and allows updating the quantities of products.

It is implemented as FormView because it's easier to adapt a FormView to display a product then adapt a DetailView to handle form validation.

Recipes

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

Customisation

How to customise models

This How-to describes how to replace Oscar models with your own. This allows you to add fields and custom methods. It builds upon the steps described in *Customising Oscar*. Please read it first and ensure that you've:

- Created a Python module with the the same app label
- Added it as Django app to `INSTALLED_APPS`
- Added a `models.py` and `admin.py`

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.abstract_models.AbstractProduct`* 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.

Tip: Using `from ... import *` is strange isn't it? Yes it is, but it needs to be done at the bottom of the module due to the way Django registers models. The order that model classes are imported makes a difference, with only the first one for a given class name being registered.

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 using migrations for this (internally Oscar already does this) so all you need to do is create a new schema migration.

It is possible to simply create a new catalogue migration (using `./manage.py makemigrations catalogue`) but this isn't recommended as any dependencies between migrations will need to be applied manually (by adding a `dependencies` attribute to the migration class).

The recommended way to handle migrations is to copy the `migrations` directory from `oscar/apps/catalogue` into your new `catalogue` app. Then you can create a new (additional) migration using the `makemigrations` management command:

```
./manage.py makemigrations catalogue
```

which will pick up any customisations to the product model.

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.

Customising dashboard forms

For example, we have customised Product model and have added several fields. And we want to show it in the form for editing. You can customise dashboard forms by creating your own form that subclasses Oscar's dashboard form for any model. For example, you can customise the `Product` form in `apps/dashboard/catalogue/forms.py` as follows:

```
from oscar.apps.dashboard.catalogue import forms as base_forms

class ProductForm(base_forms.ProductForm):

    class Meta(base_forms.ProductForm.Meta):

        fields = (
            'title', 'upc', 'on_sale',
            'short_description', 'description',
            'out_of_stock', 'bestseller',
            'is_new', 'is_discountable', 'structure',
            'markdown', 'markdown_reason')
```

Finally, make sure that you have overridden the dashboard app in your settings: `get_core_apps(['apps.dashboard.catalogue'])`.

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:

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

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            location('templates'), # templates directory of the project
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                ...
                'oscar.core.context_processors.metadata',
            ],
        },
    },
]
```

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:

```
import os
from oscar import OSCAR_MAIN_TEMPLATE_DIR
location = lambda x: os.path.join(os.path.dirname(os.path.realpath(__file__)), '..',␣
↪x)

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            location('templates'), # templates directory of the project
            OSCAR_MAIN_TEMPLATE_DIR,
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                ...
                'oscar.core.context_processors.metadata',
            ],
        },
    },
]
```

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/detail-for-upc-%s.html` or `catalogue/detail-for-class-%s.html`, where `%s` is the product's UPC or class's slug, respectively.

Example: Changing the analytics package

Suppose 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 or feature

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 = [
    ...
    url(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 disable Oscar feature

You can add feature name to the setting `OSCAR_HIDDEN_FEATURES` and its application URLs would be excluded from the URLconf. Template code, wrapped with the `{% iffeature %}{% endiffeature %}` block template tag, will not be rendered:

```
{% iffeature "reviews" %}
    {% include "catalogue/reviews/partials/review_stars.html" %}
{% endiffeature %}
```

Currently supported "reviews" and "wishlists" features. You can make your custom feature hidable by setting `hidable_feature_name` property of the `Application` class:

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

```
class LotterApplication(Application):
    hidable_feature_name = 'lottery'
```

Then, it needs to be added to the corresponding setting: `OSCAR_HIDDEN_FEATURES = ['lottery']`. Finally, you can wrap necessary template code with the `{% iffeature "lottery" %}{% endiffeature %}` tag as in the example above.

How to add views or change URLs or permissions

Oscar has many views and associated URLs. Often you want to customise these URLs for your domain, or add additional views to an app.

This how-to describes how to do just that. It builds upon the steps described in *Customising Oscar*. Please read it first and ensure that you've:

- Created a Python module with the the same label
- Added it as Django app to `INSTALLED_APPS`
- Added a `models.py` and `admin.py`

The application class

Each Oscar app comes with an application instance which inherits from *oscar.core.application.Application*. They're mainly used to gather URLs (with the correct permissions) for each Oscar app. This structure makes Oscar apps more modular as each app is responsible for its own URLs. And as it is a class, it can be overridden like any other Oscar class; hence making it straightforward to change URLs or add new views. Each app instance exposes a `urls` property, which is used to access the list of URLs of an app.

The application tree

Oscar's app instances are organised in a tree structure. The root application illustrates this nicely:

```
# oscar/app.py
class Shop(Application):
    name = None

    catalogue_app = get_class('catalogue.app', 'application')
    basket_app = get_class('basket.app', 'application')
    # ...

    def get_urls(self):
        urls = [
            url(r'^catalogue/', include(self.catalogue_app.urls)),
            url(r'^basket/', include(self.basket_app.urls)),
            # ...
        ]
```

The root app pulls in the URLs from its children. That means to add all Oscar URLs to your Django project, you only need to include the `urls` property from the root app:

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

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

Changing sub app

Sub-apps such as the `catalogue` app are loaded dynamically, just as most other classes in Oscar:

```
# oscar/app.py
class Shop(Application):
    name = None

    catalogue_app = get_class('catalogue.app', 'application')
    customer_app = get_class('customer.app', 'application')
    # ...
```

That means you just need to create another `application` instance. It will usually inherit from Oscar's version. Say you'd want to add another view to the promotions app. You only need to create a class called `PromotionsApplication` (and usually inherit from Oscar's version) and add your view:

```
# yourproject/promotions/app.py

from oscar.apps.promotions.app import PromotionsApplication as␣
↪CorePromotionsApplication
from .views import MyExtraView

class PromotionsApplication(CorePromotionsApplication):
    extra_view = MyExtraView

application = PromotionsApplication()
```

Changing the root app

If you want to e.g. change the URL for the catalogue app from `/catalogue` to `/catalog`, you need to use a custom root app instance instead of Oscar's default instance. Hence, 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 = [
            url(r'^catalog/', include(self.catalogue_app.urls)),
            # all the remaining URLs, removed for simplicity
            # ...
        ]
        return urlpatterns

application = MyShop()
```

As the root app is hardcoded in your project's `urls.py`, you need to modify it to use your new application instance instead of Oscar's default:

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

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

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

How to customise an existing view

Oscar has many views. This How-to describes how to customise one of them for your project. It builds upon the steps described in *Customising Oscar*. Please read it first and ensure that you've:

- Created a Python module with the the same label
- Added it as Django app to `INSTALLED_APPS`
- Added a `models.py` and `admin.py`

Example

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. As long as the view has the same name as the view you're replacing, and is in an app with the same name, it will get picked up automatically by Oscar.

If you want to change the template, 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>
```

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. If you add to the navigation non-dashboard URLconf, you need to set `access_fn` parameter for the current node, so that Oscar is able to resolve permissions to the current node:

```
OSCAR_DASHBOARD_NAVIGATION += [
    {
        'label': _('Admin site'),
        'icon': 'icon-dashboard',
        'url_name': 'admin:index',
        'access_fn': lambda user, url_name, url_args, url_kwargs: user.is_staff,
    }
]
```

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 add 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 restricted 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.

Controlling visibility per user

By setting `'access_fn'` for a node, you can specify a function that will get called with the current user. The node will only be displayed if that function returns `True`. If no `'access_fn'` is specified, `OSCAR_DASHBOARD_DEFAULT_ACCESS_FUNCTION` is used.

Customising Oscar's communications

Oscar provides the ability to customise the emails sent out to customers.

There are two main ways this can be achieved, either in code (via template files) or in the database (via Dashboard > Content > Email templates).

Communications API

First, it's important to understand a little about how the Communications API works.

Oscar has a model called a `CommunicationEventType`. When preparing an email to send out to a customer, the client code will do something like this:

```
commtype_code = 'SOME_EVENT'
context = {'customer': customer, 'something_else': 'Some more context.'}

try:
    event_type = CommunicationEventType.objects.get(code=commtype_code)
except CommunicationEventType.DoesNotExist:
    messages = CommunicationEventType.objects.get_and_render(commtype_code, ctx)
else:
    messages = event_type.get_messages(ctx)
```

What's happening here is:

- The code defines an arbitrary communication type code to be treated as the reference for this particular type of communication. For example, the communication type code used when sending an order email confirmation is `'ORDER_PLACED'`.
- The database is checked for a `CommunicationEventType` with this communication type code. If it does, it renders the messages using that model instance, passing in some context.
- Otherwise, it uses the `get_and_render()` method to render the messages, which uses templates instead.

So, your first step when customising the emails sent out is to work out what communication type code is being used to send out the email. The easiest way to work this out is usually to look through the email templates in `templates/oscar/customer/emails`: if the email template is called, say, `commtype_order_placed_body.html`, then the code will be `'ORDER_PLACED'`. See 'Customising through code' below.

Customising through code

Customising emails through code uses Django's standard template inheritance.

The first step is to locate the template for the particular email, which is usually in templates/oscar/customer/emails. Then, in a template directory that takes precedence over the oscar templates directory, copy the file and customise it. For example, to override the `templates/oscar/customer/emails/commtype_order_placed_body.html` template, create the file `customer/emails/commtype_order_placed_body.html` in your template directory.

Note that usually emails have three template files associated with them: the email subject line (`commtype_CODE_subject.txt`), the html version (`commtype_CODE_body.html`) and the text version (commtype_CODE_body.txt). Usually you will want to make sure you override BOTH the html and the text version.

Customising through code will not work if there is a template defined in the database instead (see below).

Customising through the database

Oscar provides a dashboard interface to allow admins to customise the emails.

To enable this for a particular communication event type, log in to the admin site and create a new `CommunicationEventType`. The code you use is the important thing: it needs to match the communication event code used when rendering the messages. For example, to override the order confirmation email, you need to create a `CommunicationEventType` with a code `'ORDER_PLACED'`.

Once you have created the `CommunicationEventType`, you can edit it using the (much better) dashboard interface at Dashboard > Content > Email templates.

If you have an email template defined in the database it will override any template files.

Customers

How to use a custom user model

You can specify a custom user model in the `AUTH_USER_MODEL` setting. Oscar will dynamically adjust the account profile summary view and profile editing form to use the fields from your custom model.

Before Django 1.5, the recommended technique for adding fields to users was to use a one-to-one "profile" model specified in the `AUTH_PROFILE_MODULE`. While this setting was removed from Django in Django 1.7, Oscar continues to support it and will add relevant fields to the profile form. Hence profiles can be used in combination with custom user models. That doesn't mean it's a good idea.

Restrictions

Oscar does have some requirements on what fields a user model has. For instance, the auth backend requires a user to have an 'email' and 'password' field. Oscar also assumes that the `email` field is unique, as this is used to identify users.

Oscar ships with its own abstract user model that supports the minimum fields and methods required for Oscar to work correctly. New Oscar projects are encouraged to subclass this User model.

Migrations

It has previously been suggested to set `db_table` of the model to `auth_user` to avoid the migrations from breaking. This issue has been fixed and migrations are now using `AUTH_USER_MODEL` and `AUTH_USER_MODEL_NAME` which will use `db_table` name of the user model provided by `get_user_model()`.

This works in the instances where you are using the default `auth.User` model or when you use a custom user model from the start. Switching over from `auth.User` to a custom model after having applied previous migration of Oscar will most likely require renaming the `auth_user` table to the new user table in a manual migration.

Example

If you want to use `oscar.apps.customer.abstract_model.AbstractUser` which has `email` as an index, and want to customize some of the methods on `User` model, say, `get_full_name` for Asian names, a simple approach is to create your own `user` module:

```
# file: your-project/apps/user/models.py
from django.db import models

from oscar.apps.customer.abstract_models import AbstractUser


class User(AbstractUser):

    def get_full_name(self):
        full_name = '%s %s' % (self.last_name.upper(), self.first_name)
        return full_name.strip()
```

Then add this `user` app to the `INSTALLED_APPS` list. Beside that we need to tell `django` to use our customized user model instead of the default one as the authentication model[1]:

```
# use our own user model
AUTH_USER_MODEL = "user.User"
```

After the migration, a database table called `user_user` will be created based on the schema defined inside of `oscar.apps.customer.abstract_models.AbstractUser`.

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',
)
```

[1] https://docs.djangoproject.com/en/stable/ref/settings/#auth-user-model

```
for breadcrumbs in categories:
    create_from_breadcrumbs(breadcrumbs)
```

Importing a catalogue

Warning: Handling imports works in Oscar, but the code quality of the importer is low as it is only used to populate the sandbox site, and not meant for general usage. So proceed at your own risk!

Importing a catalogue is pretty straightforward, and can be done in two easy steps:

- Reading the catalogue CSV file, line by line, using `UnicodeCSVReader`. `oscar.core.compat.UnicodeCSVReader` is a Unicode compatible wrapper for CSV reader and writer that abstracts away differences between Python 2 and 3.
- Using the info of each line, start by creating a `Product` object using the standard Django ORM, set the product attributes, save it, and finally set its `ProductCategory`, `Partner`, and `StockRecord`.

Example

An example of that is the `CatalogueImporter` used to import catalogues for the sandbox site. The class is available under `oscar.apps.partner.importers`.

Let's take a closer look at `CatalogueImporter`:

```
class CatalogueImporter(object):
    def __init__(self, logger):
        self.logger = logger

    @atomic
    def _import(self, file_path=None):
        ....

    def _import_row(self, row_number, row, stats):
        ....
```

The two steps procedure we talked about are obvious in this example, and are implemented in `_import` and `_import_row` functions, respectively.

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 by an *availability policy* which is loaded by the *strategy class*.

To set custom availability messaging, use your own strategy class to return the appropriate availability policy. It's possible to return different availability policies depending on the user, request and product in question.

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). 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_debited` 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.UserCancelled` During many payment flows, the user is able to cancel the process. This should often be treated differently from a payment error, e.g. it might not be appropriate to offer to retry the payment.
- `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:

```
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, **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 handle US taxes

When trading in the US, taxes aren't known until the customer's shipping address has been entered. This scenario requires two changes from core Oscar.

Ensure your site strategy returns prices without taxes applied

First, the site strategy should return all prices without tax when the customer is based in the US. Oscar provides a *US* strategy class that uses the *DeferredTax* mixin to indicate that prices don't include taxes.

See *the documentation on strategies* for further guidance on how to replace strategies.

Adjust checkout views to apply taxes once they are known

Second, the *CheckoutSessionMixin* should be overridden within your project to apply taxes to the submission.

```
from oscar.apps.checkout import session

from . import tax

class CheckoutSessionMixin(session.CheckoutSessionMixin):

    def build_submission(self, **kwargs):
        submission = super(CheckoutSessionMixin, self).build_submission(
            **kwargs)

        if submission['shipping_address'] and submission['shipping_method']:
            tax.apply_to(submission)

            # Recalculate order total to ensure we have a tax-inclusive total
            submission['order_total'] = self.get_order_totals(
                submission['basket'],
                submission['shipping_charge'])

        return submission
```

An example implementation of the `tax.py` module is:

```
from decimal import Decimal as D

def apply_to(submission):
    # Assume 7% sales tax on sales to New Jersey  You could instead use an
    # external service like Avalara to look up the appropriates taxes.
    STATE_TAX_RATES = {
        'NJ': D('0.07')
    }
    shipping_address = submission['shipping_address']
    rate = STATE_TAX_RATES.get(
        shipping_address.state, D('0.00'))
    for line in submission['basket'].all_lines():
        line_tax = calculate_tax(
            line.line_price_excl_tax_incl_discounts, rate)
        unit_tax = (line_tax / line.quantity).quantize(D('0.01'))
        line.purchase_info.price.tax = unit_tax

    # Note, we change the submission in place - we don't need to
    # return anything from this function
    shipping_charge = submission['shipping_charge']
    if shipping_charge is not None:
        shipping_charge.tax = calculate_tax(
            shipping_charge.excl_tax, rate)
```

```
def calculate_tax(price, rate):
    tax = price * rate
    return tax.quantize(D('0.01'))
```

Shipping

How to configure 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

Configuring shipping charges requires overriding Oscar's core 'shipping' app and providing your own `Repository` class (see *Customising Oscar*) that returns your chosen shipping method instances.

The primary responsibility of the `Repository` class is to provide the available shipping methods for a particular scenario. This is done via the *`get_shipping_methods()`* method, which returns the shipping methods available to the customer.

This method is called in several places:

- To look up a "default" shipping method so that sample shipping charges can be shown on the basket detail page.
- To list the available shipping methods on the checkout shipping method page.
- To check the selected shipping method is still available when an order is submitted.

The `get_shipping_methods` method takes the basket, user, shipping address and request as parameters. These can be used to provide different sets of shipping methods depending on the circumstances. For instance, you could use the shipping address to provide international shipping rates if the address is overseas.

The `get_default_shipping_method` method takes the same parameters and returns default shipping method for the current basket. Used for shipping cost indication on the basket page. Defaults to free shipping method.

Note: Oscar's checkout process includes a page for choosing your shipping method. If there is only one method available for your basket (as is the default) then it will be chosen automatically and the user immediately redirected to the next step.

Custom repositories

If the available shipping methods are the same for all customers and shipping addresses, then override the `methods` property of the repository:

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

class Repository(repository.Repository):
    methods = (methods.Standard(), methods.Express())
```

For more complex logic, override the `get_available_shipping_methods` method:

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

class Repository(repository.Repository):

    def get_available_shipping_methods(
            self, basket, user=None, shipping_addr=None,
            request=None, **kwargs):
        methods = (methods.Standard())
        if shipping_addr and shipping_addr.country.code == 'GB':
            # Express is only available in the UK
            methods = (methods.Standard(), methods.Express())
        return methods
```

Note that the `get_shipping_methods` method wraps `get_available_shipping_methods` in order to handle baskets that don't require shipping and to apply shipping discounts.

Shipping methods

Shipping methods need to implement a certain API. They need to have the following properties which define the metadata about the shipping method:

- `code` - This is used as an identifier for the shipping method and so should be unique amongst the shipping methods available in your shop.
- `name` - The name of the shipping method. This will be visible to the customer during checkout.
- `description` - An optional description of the shipping method. This can contain HTML.

Further, each method must implement a `calculate` method which accepts the basket instance as a parameter and returns a `Price` instance. Most shipping methods subclass *`Base`*, which stubs this API.

Here's an example:

```
from oscar.apps.shipping import methods
from oscar.core import prices

class Standard(methods.Base):
    code = 'standard'
    name = 'Standard shipping (free)'

    def calculate(self, basket):
        return prices.Price(
            currency=basket.currency,
            excl_tax=D('0.00'), incl_tax=D('0.00'))
```

Core shipping methods

Oscar ships with several re-usable shipping methods which can be used as-is, or subclassed and customised:

- *Free* - no shipping charges
- *FixedPrice* - fixed-price shipping charges. Example usage:

```
from oscar.apps.shipping import methods
from oscar.core import prices

class Standard(methods.FixedPrice):
    code = 'standard'
    name = 'Standard shipping'
    charge_excl_tax = D('5.00')

class Express(methods.FixedPrice):
    code = 'express'
    name = 'Express shipping'
    charge_excl_tax = D('10.00')
```

There is also a weight-based shipping method, `AbstractWeightBased` which determines a shipping charge by calculating the weight of a basket's contents and looking this up in a model-based set of weight bands.

Order processing

How to set up order processing

How orders are processed differs for every shop. Some shops will process orders manually, using the dashboard to print picking slips and update orders once items have shipped. Others will use automated processes to send order details to fulfillment partners and pick up shipment and cancellation messages.

Oscar provides only a skeleton for building your order processing pipeline on top of. This page details how it works and how to build your order processing pipeline.

Structure

There are two relevant Oscar apps to order processing.

- The checkout app is responsible for collecting the required shipping and payment information, taking payment in some sense and placing the order. It is not normally used to process the order in any sense. If your orders can be fulfilled immediately after being placed (eg digital products), it's better to use a separate process (like a cronjob or celery task). That way, if the fulfilment work fails for some reason, it can be retried easily later. It's also a neater decoupling of responsibilities.
- The order app has a `processing.py` module which is intended to handle order processing tasks, such as items being cancelled, shipped or returned. More details below.

Modelling

Oscar models order processsing through events. There are three types to be aware of:

- Shipping events. These correspond to some change in the location or fulfilment status of the order items. For instance, when items are shipped, returned or cancelled. For digital goods, this would cover when items are downloaded.

- Payment events. These model each transaction that relates to an order. The payment model allows order lines to be linked to the payment event.
- Communication events. These capture emails and other messages sent to the customer about a particular order. These aren't a core part of order processing and are used more for audit and to ensure, for example, that only one order confirmation email is sent to a customer.

Event handling

Most Oscar shops will want to customise the `EventHandler` class from the order app. This class is intended to handle all events and perform the appropriate actions. The main public API is

class `oscar.apps.order.processing.`**`EventHandler`**(*user=None*)
Handle requested order events.

This is an important class: it houses the core logic of your shop's order processing pipeline.

`handle_order_status_change`(*order*, *new_status*, *note_msg=None*)
Handle a requested order status change

This method is not normally called directly by client code. The main use-case is when an order is cancelled, which in some ways could be viewed as a shipping event affecting all lines.

`handle_payment_event`(*order*, *event_type*, *amount*, *lines=None*, *line_quantities=None*, ***kwargs*)
Handle a payment event for a given order.

These should normally be called as part of handling a shipping event. It is rare to call to this method directly. It does make sense for refunds though where the payment event may be unrelated to a particular shipping event and doesn't directly correspond to a set of lines.

`handle_shipping_event`(*order*, *event_type*, *lines*, *line_quantities*, ***kwargs*)
Handle a shipping event for a given order.

This is most common entry point to this class - most of your order processing should be modelled around shipping events. Shipping events can be used to trigger payment and communication events.

You will generally want to override this method to implement the specifics of you order processing pipeline.

Many helper methods are also provided:

class `oscar.apps.order.processing.`**`EventHandler`**(*user=None*)
Handle requested order events.

This is an important class: it houses the core logic of your shop's order processing pipeline.

`are_stock_allocations_available`(*lines*, *line_quantities*)
Check whether stock records still have enough stock to honour the requested allocations.

`calculate_payment_event_subtotal`(*event_type*, *lines*, *line_quantities*)
Calculate the total charge for the passed event type, lines and line quantities.

This takes into account the previous prices that have been charged for this event.

Note that shipping is not including in this subtotal. You need to subclass and extend this method if you want to include shipping costs.

`cancel_stock_allocations`(*order*, *lines*, *line_quantities*)
Cancel the stock allocations for the passed lines

`consume_stock_allocations`(*order*, *lines*, *line_quantities*)
Consume the stock allocations for the passed lines

have_lines_passed_shipping_event(*order*, *lines*, *line_quantities*, *event_type*)
Test whether the passed lines and quantities have been through the specified shipping event.

This is useful for validating if certain shipping events are allowed (ie you can't return something before it has shipped).

validate_shipping_event(*order*, *event_type*, *lines*, *line_quantities*, ***kwargs*)
Test if the requested shipping event is permitted.

If not, raise InvalidShippingEvent

Most shops can handle all their order processing through shipping events, which may indirectly create payment events.

Customisation

Assuming your order processing involves an admin using the dashboard, then the normal customisation steps are as follows:

1. Ensure your orders are created with the right default status.
2. Override the order dashboard's views and templates to provide the right interface for admins to update orders.
3. Extend the `EventHandler` class to correctly handle shipping and payment events that are called from the dashboard order detail view.

It can be useful to use order and line statuses too. Oscar provides some helper methods to make this easier.

class `oscar.apps.order.abstract_models.`**AbstractOrder**(**args*, ***kwargs*)
The main order model

classmethod all_statuses()
Return all possible statuses for an order

available_statuses()
Return all possible statuses that this order can move to

pipeline = {'Cancelled': (), 'Being processed': ('Complete', 'Cancelled'), 'Pending': ('Being processed', 'Cancelled'), '
Order status pipeline. This should be a dict where each (key, value) # – corresponds to a status and a list of possible statuses that can follow that one.

set_status(*new_status*)
Set a new status for this order.

If the requested status is not valid, then `InvalidOrderStatus` is raised.

class `oscar.apps.order.abstract_models.`**AbstractLine**(**args*, ***kwargs*)
An order line

classmethod all_statuses()
Return all possible statuses for an order line

available_statuses()
Return all possible statuses that this order line can move to

pipeline = {}
Order status pipeline. This should be a dict where each (key, value) corresponds to a status and the possible statuses that can follow that one.

set_status(*new_status*)
Set a new status for this line

If the requested status is not valid, then `InvalidLineStatus` is raised.

Example

Here is a reasonably common scenario for order processing. Note that some of the functionality described here is not in Oscar. However, Oscar provides the hook points to make implementing this workflow easy.

- An order is placed and the customer's bankcard is pre-authed for the order total. The new order has status 'Pending'
- An admin logs into the dashboard and views all new orders. They select the new order, retrieve the goods from the warehouse and get them ready to ship.
- When all items are retrieved, they select all the lines from the order and hit a button saying 'take payment'. This calls the `handle_payment_event` method of the `EventHandler` class which performs the settle transaction with the payment gateway and, if successful, creates a payment event against the order.
- If payment is successful, the admin ships the goods and gets a tracking number from the courier service. They then select the shipped lines for the order and hit a button saying "mark as shipped". This will show a form requesting a shipping number for the event. When this is entered, the `handle_shipping_event` method of the `EventHandler` class is called, which will update stock allocations and create a shipping event.

Offers

How to create a custom range

Oscar ships with a range model that represents a set of products from your catalogue. Using the dashboard, 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.
- have an `all_products` method that returns a queryset of all products in the range.

Example:

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

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

    def num_products(self):
        return self.all_products().count()

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

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 `description` attribute which describes what needs to happen to satisfy the condition (eg "basket must have 4 items").
- have an `is_satisfied` method that takes a basket instance and an offer instance and returns a boolean indicating if the condition is satisfied

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 upsell 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, offer, 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_condition(BasketOwnerCalledBarry)
```

Now you should see this condition 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 be 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):
        """
        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, order, application):
        """
        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):
        # 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, order, application):
        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"
```

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 the Bootstrap project. Classes for styling should be separate from classes used for Javascript. The latter must be prefixed with `js-`, and using data attributes is often preferable.

Frontend vs. Dashboard

The frontend and dashboard are intentionally kept very separate. They incidentally both use Bootstrap, but may be updated individually. The frontend is based on Bootstrap's LESS files and ties it together with Oscar-specific styling in `styles.less`.

On the other hand, `dashboard.less` just contains a few customisations that are included alongside a copy of stock Bootstrap CSS - and at the time of writing, using a different Bootstrap version.

LESS/CSS

By default, CSS files compiled from their LESS sources are used rather than the LESS ones. To use Less directly, set `OSCAR_USE_LESS = True` in your settings file. This will enable browser LESS pre-processor which lets you trial changes with a page reload. If you want to commit your changes, use the `make css` Makefile command, which uses *Gulp* for compiling into CSS. A few other CSS files are used to provide styles for javascript libraries.

Javascript

Oscar uses javascript for progressive enhancements. This guide used to document exact versions, but quickly became outdated. It is recommended to inspect `layout.html` and `dashboard/layout.html` for what is currently included.

Customisation

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 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 %}
```

Deployment and setup

How to setup Solr with Oscar

Apache Solr is Oscar's recommended production-grade search backend. This how-to describes how to get Solr running, and integrated with Oscar. The instructions below are tested on an Ubuntu machine, but should be applicable for similar environments. A working Java or OpenJDK installation are necessary.

Installing Solr

You first need to fetch and extract Solr. The schema included with Oscar is tested with Solr 4.7.2:

```
$ wget http://archive.apache.org/dist/lucene/solr/4.7.2/solr-4.7.2.tgz
$ tar xzf solr-4.7.2.tgz
```

Integrating with Haystack

Haystack provides an abstraction layer on top of different search backends and integrates with Django. Your Haystack connection settings in your `settings.py` for the config above should look like this:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://127.0.0.1:8983/solr',
        'INCLUDE_SPELLING': True,
    },
}
```

Build solr schema

Next, replace the example configuration with Oscar's.

```
$ ./manage.py build_solr_schema > solr-4.7.2/example/solr/collection1/conf/schema.xml
```

You should then be able to start Solr by running:

```
$ cd solr-4.7.2/example
$ java -jar start.jar
```

Rebuild search index

If all is well, you should now be able to rebuild the search index.

```
$ ./manage.py rebuild_index --noinput
Removing all documents from your index because you said so.
All documents removed.
Indexing 201 Products
Indexing 201 Products
```

The products being indexed twice is caused by a low-priority bug in Oscar and can be safely ignored. If the indexing succeeded, search in Oscar will be working. Search for any term in the search box on your Oscar site, and you should get results.

Oscar settings

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

Display settings

OSCAR_SHOP_NAME

Default: `'Oscar'`

The name of your e-commerce shop site. This is shown as the main logo within the default templates.

OSCAR_SHOP_TAGLINE

Default: `''`

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

OSCAR_HOMEPAGE

Default: `reverse_lazy('promotions:home')`

URL of home page of your site. This value is used for *Home* link in navigation and redirection page after logout. Useful if you use a different app to serve your homepage.

OSCAR_ACCOUNTS_REDIRECT_URL

Default: `'customer:profile-view'`

Oscar has a view that gets called any time the user clicks on 'My account' or similar. By default it's a dumb redirect to the view configured with this setting. But you could also override the view to display a more useful account summary page or such like.

OSCAR_RECENTLY_VIEWED_PRODUCTS

Default: 20

The number of recently viewed products to store.

OSCAR_RECENTLY_VIEWED_COOKIE_LIFETIME

Default: 604800 (1 week in seconds)

The time to live for the cookie in seconds.

OSCAR_RECENTLY_VIEWED_COOKIE_NAME

Default: `'oscar_history'`

The name of the cookie for showing recently viewed products.

OSCAR_HIDDEN_FEATURES

Defaults: `[]`

Allows to disable particular Oscar feature in application and templates. More information in the *How to disable an app or feature* document.

Pagination

There are a number of settings that control pagination in Oscar's views. They all default to 20.

- OSCAR_PRODUCTS_PER_PAGE
- OSCAR_OFFERS_PER_PAGE
- OSCAR_REVIEWS_PER_PAGE
- OSCAR_NOTIFICATIONS_PER_PAGE
- OSCAR_EMAILS_PER_PAGE
- OSCAR_ORDERS_PER_PAGE
- OSCAR_ADDRESSES_PER_PAGE
- OSCAR_STOCK_ALERTS_PER_PAGE
- OSCAR_DASHBOARD_ITEMS_PER_PAGE

OSCAR_SEARCH_FACETS

A dictionary that specifies the facets to use with the search backend. It needs to be a dict with keys `fields` and `queries` for field- and query-type facets. Field-type facets can get an 'options' element with parameters like facet sorting, filtering, etc. The default is:

```
OSCAR_SEARCH_FACETS = {
    'fields': OrderedDict([
        ('product_class', {'name': _('Type'), 'field': 'product_class'}),
        ('rating', {'name': _('Rating'), 'field': 'rating'}),
    ]),
    'queries': OrderedDict([
        ('price_range',
         {
             'name': _('Price range'),
             'field': 'price',
             'queries': [
                 # This is a list of (name, query) tuples where the name will
                 # be displayed on the front-end.
                 (_('0 to 20'), u'[0 TO 20]'),
                 (_('20 to 40'), u'[20 TO 40]'),
                 (_('40 to 60'), u'[40 TO 60]'),
```

```
                (_('60+'), u'[60 TO *]'),
            ]
        }),
    ]),
}
```

OSCAR_PRODUCT_SEARCH_HANDLER

The search handler to be used in the product list views. If `None`, Oscar tries to guess the correct handler based on your Haystack settings.

Default:

```
None
```

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. Usage is explained in *How to configure the dashboard navigation*.

OSCAR_DASHBOARD_DEFAULT_ACCESS_FUNCTION

Default: `'oscar.apps.dashboard.nav.default_access_fn'`

OSCAR_DASHBOARD_NAVIGATION allows passing an access function for each node which is used to determine whether to show the node for a specific user or not. If no access function is defined, the function specified here is used. The default function integrates with the permission-based dashboard and shows the node if the user will be able to access it. That should be sufficient for most cases.

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_LINE_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.

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', 'line4', 'postcode', 'country')`

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

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.

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.

Offer settings

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.

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'`

The name of the cookie for the open basket.

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_FORMAT

Default: `None`

Dictionary with arguments for the `format_currency` function from the Babel library. Contains next options: *format*, *format_type*, *currency_digits*. For example:

```
OSCAR_CURRENCY_FORMAT = {
    'USD': {
        'currency_digits': False,
        'format_type': "accounting",
    },
    'EUR': {
        'format': u'#,##0\xa0¤',
    }
}
```

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_DELETE_IMAGE_FILES

Default: `True`

If enabled, a `post_delete` hook will attempt to delete any image files and created thumbnails when a model with an `ImageField` is deleted. This is usually desired, but might not be what you want when using a remote storage.

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.

Slug settings

OSCAR_SLUG_MAP

Default: `{}`

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#': 'fsharp',
}
```

OSCAR_SLUG_FUNCTION

Default: `'oscar.core.utils.default_slugifier'`

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. String notation is recommended, but specifying a callable is supported for backwards-compatibility.

Example:

```
# in myproject.utils
def some_slugify(value):
    return value

# in settings.py
OSCAR_SLUG_FUNCTION = 'myproject.utils.some_slugify'
```

OSCAR_SLUG_BLACKLIST

Default: `[]`

A list of words to exclude from slugs.

Example:

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

OSCAR_SLUG_ALLOW_UNICODE

Default: `False`

Allows unicode characters in slugs generated by `AutoSlugField`, which is supported by the underlying `SlugField` in Django>=1.9.

Misc settings

OSCAR_COOKIES_DELETE_ON_LOGOUT

Default: `['oscar_recently_viewed_products',]`

Which cookies to delete automatically when the user logs out.

OSCAR_GOOGLE_ANALYTICS_ID

Tracking ID for Google Analytics tracking code, available as *google_analytics_id* in the template context. If setting is set, enables Universal Analytics tracking code for page views and transactions.

OSCAR_USE_LESS

Allows to use raw LESS styles directly. Refer to *LESS/CSS* document for more details.

OSCAR_CSV_INCLUDE_BOM

Default: `False`

A flag to control whether Oscar's CSV writer should prepend a byte order mark (BOM) to CSV files that are encoded in UTF-8. Useful for compatibility with some CSV readers, Microsoft Excel in particular.

Signals

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

product_viewed

class `oscar.apps.catalogue.signals.`**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

product_search

class oscar.apps.catalogue.signals.**product_search**
Raised when a search is performed.

Arguments sent with this signal:

query
The search term

user
The user in question

user_registered

class oscar.apps.customer.signals.**user_registered**
Raised when a user registers

Arguments sent with this signal:

request
The request instance

user
The user in question

basket_addition

class oscar.apps.basket.signals.**basket_addition**
Raised when a product is added to a basket

Arguments sent with this signal:

request
The request instance

product
The product being added

user
The user in question

voucher_addition

class oscar.apps.basket.signals.**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

start_checkout

class oscar.apps.checkout.signals.**start_checkout**
Raised when the customer begins the checkout process

Arguments sent with this signal:

request
The reuqest instance

pre_payment

class oscar.apps.checkout.signals.**pre_payment**
Raised immediately before attempting to take payment in the checkout.

Arguments sent with this signal:

view
The view class instance

post_payment

class oscar.apps.checkout.signals.**post_payment**
Raised immediately after payment has been taken.

Arguments sent with this signal:

view
The view class instance

order_placed

class oscar.apps.order.signals.**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!)

post_checkout

class oscar.apps.checkout.signals.**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

review_created

class oscar.apps.catalogue.reviews.signals.**review_added**
Raised when a review is added.

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

Template tags

Shipping tags

Load these tags using `{% load shipping_tags %}`.

shipping_charge

Injects the shipping charge into the template context:

Usage:

```
{% shipping_charge shipping_method basket as name %}
Shipping charge is {{ name }}.
```

The arguments are:

Argument	Description
shipping_method	The shipping method instance
basket	The basket instance to calculate shipping charges for
name	The variable name to assign the charge to

shipping_charge_discount

Injects the shipping discount into the template context:

Usage:

```
{% shipping_discount shipping_method basket as name %}
Shipping discount is {{ charge }}.
```

The arguments are:

Argument	Description
shipping_method	The shipping method instance
basket	The basket instance to calculate shipping charges for
name	The variable name to assign the charge to

shipping_charge_excl_discount

Injects the shipping charges with no discounts applied into the template context:

Usage:

```
{% shipping_charge_excl_discount shipping_method basket as name %}
Shipping discount is {{ name }}.
```

The arguments are:

Argument	Description
shipping_method	The shipping method instance
basket	The basket instance to calculate shipping charges for
name	The variable name to assign the charge to

The Oscar open-source project

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

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:

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 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.

Classes are loaded dynamically

The complexity of scenarios doesn't stop with Django models; core parts of Oscar need to be as customisable as possible. Hence almost all classes (including views) are *dynamically loaded*, which results in a maintainable approach to customising behaviour.

URLs and permissions for apps are handled by Application instances

The `oscar.core.application.Application` class handles mapping URLs to views and permissions at an per-app level. This makes Oscar's apps more modular, and makes it easy to customise this mapping as they can be overridden just like any other class in Oscar.

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.

Release notes

Release notes for each version of Oscar published to PyPI.

1.5 release branch

Oscar 1.5 release notes

release 2017-08-10

Welcome to Oscar 1.5

Table of contents:

Compatibility

Oscar 1.5 is compatible with Django 1.8, 1.10 and 1.11 as well as Python 2.7, 3.3, 3.4, 3.5 and 3.6. Support for Django 1.9 is no longer officialy supported since it is longer supported by Django (end of life).

What's new in Oscar 1.5?

- Support for Django 1.11 (#2221)
- Google Analytics tracking code has been migrated to Universal Analytics.

- Passwords are validated using the validators defined in `AUTH_PASSWORD_VALIDATORS` for Django 1.9 and above.
- The custom PhoneNumberField implementation has been replaced with the django-phonenumber-field package (#2227)

Removal of deprecated features

These methods/modules have been removed:

- Profiling middleware. See silk or django-cprofile-middleware for alternatives.
- `Product.min_child_price_incl_tax` and `Product.min_child_price_excl_tax`.
- `oscar.core.logging.handlers.EnvFileHandler()`
- The `ellipses_page_range` templatetag. See django-rangepaginator for an alternative.
- `oscar.test.decorators` module.
- `oscar.core.utils.compose` function.
- `oscar.apps.customer.auth_backends.Emailbackend`. Use `oscar.apps.customer.auth_backends.EmailBackend` instead.
- `oscar.apps.catalogue.search_handlers.ProductSearchHandler`. Use
- `oscar.apps.catalogue.search_handlers.SolrProductSearchHandler` instead.

Minor changes

- Added billing address to user's address book during checkout (#1532). Number of usages for billing and shipping addresses tracked separately: billing address in `UserAddress.num_orders_as_billing_address` field and shipping address in `UserAddress.num_order_as_shipping_address` accordingly.
- Fixed logic for determining "hurry mode" on stock alerts. Hurry mode is now set if the number of alerts for a product exceeds the quantity in stock, rather than the other way around.
- Updated search to use the haystack CharField instead of the EdgeNgramField to fix irrelevant results (#2128)
- Updated `customer.utils.Dispatcher` to improve the ability to customise the implementation (#2303)
- Fixed logic for determining "hurry mode" on stock alerts. Hurry mode is now set if the number of alerts for a product exceeds the quantity in stock, rather than the other way around.
- `Voucher.date_created` and `VoucherApplication.date_created` model fields changed from `DateField` to `DateTimeField`. If you use SQLite database, please make sure to backup data before running migrations, since due to limited support of column altering, Django re-creates new SQLite database with new columns and copies into it original data - https://docs.djangoproject.com/en/1.11/topics/migrations/#sqlite
- Allowed the communication event dispatcher to accept an optional `mail_connection` to use when sending email messages.
- Changed product alerts to be sent using the Communication Event framework. The old email templates will continue to be supported until Oscar 2.0.

Backwards incompatible changes in Oscar 1.5

- Product reviews are now deleted if the associated product is deleted.
- Formset classes moved to the separate modules, if you import them directly - please update location or use `oscar.core.loading.get_classes()` for loading classes (#1957).

  Next classes have new locations:

- `BaseBasketLineFormSet`, `BasketLineFormSet`, `BaseSavedLineFormSet`, `SavedLineFormSet` moved to `oscar.apps.basket.formsets` module;
- `BaseStockRecordFormSet`, `StockRecordFormSet`, `BaseProductCategoryFormSet`, `ProductCategoryFormSet`, `BaseProductImageFormSet`, `ProductImageFormSet`, `BaseProductRecommendationFormSet`, `ProductRecommendationFormSet`, `ProductAttributesFormSet` moved to `oscar.apps.dashboard.catalogue.formsets`;
- `OrderedProductFormSet` moved to `oscar.apps.dashboard.promotions.formsets`; - `LineFormset` moved to `oscar.apps.wishlists.formsets`.
- `SimpleAddToBasketForm` doesn't override the quantity field any more. Instead, it just hides the field declared by AddToBasketForm and sets the quantity to one. This means `SimpleAddToBasketForm` doesn't need to be overridden for most cases, but please check things still work as expected for you if you have customized it.
- `OSCAR_CURRENCY_FORMAT` setting changed to dictionary form in order to support multi-currency for currency formatting. You can set *format*, *format_type* and *currency_digits* in it. Please refer to documentation for an example.
- Dashboard order list doesn't have shortcut filters any more, pass *status* parameter instead of *order_status* for the relevant filtering.
- `GOOGLE_ANALYTICS_ID` and `USE_LESS` settings were renamed into `OSCAR_GOOGLE_ANALYTICS_ID` and `OSCAR_USE_LESS` respectively in order to keep all Oscar settings under common namespace.
- Removed `display_version` and `version` variables from templates and template context.
- Offer `Applicator` is now loaded from the `offer.applicator` module, instead of `offer.utils`. Old path is deprecated and won't be supported in the next Oscar versions.
- `oscar.forms.fields.ExtendedURLField` no longer accepts a `verify_exists` argument.

Dependency changes

The following packages are updated:

- *django>=1.8.8,<1.12*
- *django-phonenumber-field>=1.0.0,<2.0.0*
- *django-haystack>=2.5.0,<2.7.0*

Oscar 1.5.1 release notes

release 2017-11-29

This is Oscar 1.5.1, a bug fix release.

Bug fixes

- Fixed django-haystack dependency to allow using the latest version of Django Haystack.

Oscar 1.5.2 release notes

release 2018-02-12

This is Oscar 1.5.2, a bug fix release.

Bug fixes

- Fixed `oscar.core.widgets.AdvancedSelect` widget behaviour with Django 1.11 and above.
- Use a local copy of TinyMCE instead of a CDN, because of recent bugs in the CDN.

Oscar 1.5.3 release notes

release 2018-04-11

This is Oscar 1.5.3, a security release.

A security vulnerability existed in the mechanism used to generate verification hashes for anonymous orders. This has been fixed in this release.

`oscar.apps.order.Order.verification_hash()` now uses `django.core.signing` instead of generating its own MD5 hash for tracking URLs for anonymous orders.

Projects that allow anonymous checkout are **strongly recommended** to generate a new `SECRET_KEY`, as the vulnerability exposed the `SECRET_KEY` to potential exposure due to weaknesses in the hash generation algorithm.

As a result of this change, order verification hashes generated previously will no longer validate by default, and URLs generated with the old hash will not be accessible.

Projects that wish to allow validation of old hashes must specify a `OSCAR_DEPRECATED_ORDER_VERIFY_KEY` setting that is equal to the `SECRET_KEY` that was in use prior to applying this change.

Oscar 1.5.4 release notes

release 2018-04-30

This is Oscar 1.5.4, a bugfix release.

Bug fixes

- Pinned the version of django-extra-views to <0.11, to avoid exceptions raised in newer versions of the package.

Dependency changes

- Upgraded TinyMCE to version 4.7.11.

1.4 release branch

Oscar 1.4 release notes

release 2017-02-10

Welcome to Oscar 1.4, a relative minor release which finally brings compatibility with Django 1.10.

Table of contents:

Compatibility

Oscar 1.4 is compatible with Django 1.8, 1.9 and 1.10 as well as Python 2.7, 3.3, 3.4, 3.5 and 3.6.

What's new in Oscar 1.4?

- Django 1.10 compatibility (#2055)
- If a product variant has no primary image then the primary image of the parent is now served. (#1998)

Minor changes

- The tracking pixel in the dashboard is removed since the domain didn't exist anymore (#2144).
- Fix sending direct messages to users (#2138)
- Remove unique constraint for proxy class field of offer condition model. (#2166)
- Change the default argument of ProductReview.status to a callable
- Use get_class for imports to partner.prices and partner.availability (#2035)
- Fix an IntegrityError when a product is moved between wishlists (#2133)
- Fix issue when update product attributes in the dashboard when the product has a related entity (#2015)
- Fix the offers report in the dashboard on Python 3 (#2223)
- It is no longer needed to pass `Application.urls` to the `include()` function when creating the urlpatterns. You can pass it directly to the `url()` function instead. (#2222)
- Oscar now set's the `on_delete` kwarg on all ForeignKey's since leaving it out is deprecated in Django 1.10+
- `oscar.core.compat.user_is_authenticated()` is now used instead of `user.is_authenticated()`, since this resulted in Deprecation warnings for Django 1.10+. (#2222)

Dependency changes

The following packages are updated:

- *treebeard* >= *4.1.0* (Django 1.10 support)

- *Pillow >= 3.4.2* (Security issue)
- *phonenumbers>=6.3.0,<9.0.0* (Allow version 8.x)
- *django-tables2 >= 1.2.0,<2.0.0*

1.3 release branch

Oscar 1.3 release notes

release 2016-08-13

Welcome to Oscar 1.3, a minor release which finally brings compatibility with Django 1.9.

Table of contents:

Compatibility

Oscar 1.3 is compatible with Django 1.8 and 1.9 as well as Python 2.7, 3.3, 3.4 and 3.5.

What's new in Oscar 1.3?

- Django 1.9 compatibility
- Update the queryset for *ProductReview* to include an *approved* method which filters the queryset on approved reviews (#1920)
- Make more use of the datepicker in the dashboard (#1983)
- Add *site* kwarg to the OrderCreator class (#2014)
- Update Oscar's *get_class()* method to work with non oscar modules (#2039)
- The ProductAttributesContainer is now moved to *catalogue.product_attribute* and can now be replaced more easily.

Dependency changes

Warning: Oscar depends on sorl-thumbnail 12.4a1. Unfortunately you have to fake the initial migration of this application since older versions didn't include migrations. This can be done via:

```
python manage.py migrate thumbnail --fake
```

The following packages are updated:

- *sorl-thumbnail == 12.4a1* (Django 1.9 support)
- *django-haystack >= 2.5.0* (Django 1.9 support)

- *mock < 3.0*
- *factory-boy > 2.4.1,< 3.0*

1.2 release branch

Oscar 1.2 release notes

release 2016-03-18

Welcome to Oscar 1.2.

Table of contents:

Compatibility

Oscar 1.2 is compatible with Django 1.7 and 1.8 as well as Python 2.7, 3.3, 3.4 and 3.5.

What's new in Oscar 1.2?

- `django-compressor` has been removed as a dependency, and as a way of building `less` files for development. Removing or disabling it was commonly required as it didn't work well with deploying on PaaS providers, and many current projects understandably prefer to use Javascript build chains (`gulp`, `grunt`, etc.) for all their statics. But `django-compressor` was hard to remove on a per-project basis, so the decision was made to remove it altogether.

  For development, `USE_LESS` now enables the browser-based on-the-fly pre-processor. `make css` continues to run a locally installed LESS binary.

Minor changes

- Fix missing page_url field in the promotions form (#1816)
- The order of basket, order and wishlist lines in now guaranteed to be in the order of creation. Previously, this wasn't guaranteed, but still almost always the case.
- *Partner* instances got a default ordering by name.
- If a child product has no weight, we check if a parent's product weight is set before falling back to the default weight (#1965).
- *Address.active_address_fields* now uses the saner common name of a country instead of the official name (#1964).
- Custom benefits now don't enforce uniqueness on the `proxy_class` field, making them more useful (#685).
- Move logic to create basket messages to it's own utility class *basket.utils.BasketMessageGenerator()*. (#1930)

- Fix a caching issue in *Product.get_absolute_url()* (#1925)
- Update the *recently_viewed_products* templatetag to accept a *current_product* attribute. (#1948)

Backwards incompatible changes in Oscar 1.2

- The `mainstyles` template block was removed. It served as a wrapper for the `styles` content black and was needed to be extensible while still being able to compress CSS. As `django-compressor` has been removed, it's not needed any more. Just use `styles` instead if you happened to use it.
- The *keywords* block is removed from the main template (#1799)
- The US and Demo sites were both removed from the repository as they were not up-to-date anymore. These might return in the future as separate repositories.
- The *RecentReviewsManager*, *TopScoredReviewsManager* and *TopVotedReviewsManager* managers are removed from the reviews app since they were broken and unused.
- A new unique index is added to *catalogue.AbstractAttributeOption* to make sure that the *group*, *option* combination is unique (#1935)

Dependency changes

- **The following packages are updated:**
  - *django-treebeard* >= *4.0* (Django 1.9 support)
  - *sorl.thumbnail* >= *12.4a1* (Django 1.9 support)
- JQuery UI is no longer included in the dashboard (#1792)

Oscar 1.2.1 release notes

release 2016-03-23

This is Oscar 1.2.1, a bug fix release.

Minor changes

Change the babel requirement in setup.py to allow versions up to 3.0

Bug fixes

- #2019: Optimize ORM query on the offer detail page in the dashboard.

Oscar 1.2.2 release notes

release 2016-07-11

This is Oscar 1.2.2, a bug fix release.

Bug fixes

- #2030: Fix migration issues on Python 3.

1.1 release branch

Oscar 1.1 release notes

release 2015-06-20

Welcome to Oscar 1.1, or the "Bootstrap 3" release. We also squashed many bugs that were reported in the last seven months, and managed to shed a lot of compatibility helpers when Django 1.6 support was removed.

Table of contents:

Compatibility

Oscar 1.1 is compatible with Django 1.7 and 1.8 as well as Python 2.7, 3.3 and 3.4. Support for Django 1.6, and hence South for migrations, has been removed.

What's new in Oscar 1.1?

- The frontend and backend templates have been updated to use Bootstrap 3 instead of version 2 (#1576). The frontend and backend templates are now also independent of each other.
- Category slug, name and URL handling has been refactored to make it easier to translate categories, and to be able to edit slugs independent of names.
- The icon and caption of *django-tables2* tables can be set directly on the *Table* object, if it derives from `DashboardTable`. The caption can be localized in singular and plural. (#1482)
- Oscar now ships with basic ElasticSearch support. `OSCAR_PRODUCT_SEARCH_HANDLER` has been introduced to more easily set the search backend.
- The offer models can now also be customised the same way as was already possible for the other apps.
- The test suite is now run with the glorious py.test.

Minor changes

- The *Order.date_placed* field can now be set explicitly rather than using the *auto_now_add* behaviour (#1558).
- The settings `OSCAR_BASKET_COOKIE_SECURE` and `OSCAR_RECENTLY_VIEWED_COOKIE_SECURE` are introduced to set the `secure` flag on the relevant cookies.

- Previously, all views handled in Oscar's Application class were decorated with the permissions_required decorator. That decorator would not do anything if used with an empty set of permissions. But it was raised as an issue, and now views not requiring permissions are not decorated at all.
- Properly redirect users to the checkout page after a guest user created an account.
- `OSCAR_SEARCH_FACETS` now accepts ordered dicts.
- Oscar now supports varying the tax rate per product.
- Product class options and attributes can now be edited in the dashboard.
- All modelforms now specify the *fields* meta attribute instead of the *excludes* list.

Backwards incompatible changes in 1.1

Categories refactor

The Category model contained two denormalisations to improve performance: it stored the name of the category and it's ancestors in `full_name`, and the `slug` field did not just contain the category's slug, but also the ones of its ancestors.

This came with several drawbacks: it was fiddly to move and update categories, as one had to ensure to update parts of the entire category tree. It also made it trickier to add model-level translation to categories.

A refactoring removed the denormalisations leading to much simpler logic, and a light sprinkle of caching for the URLs hopefully leads to a performance net positive. But unfortunately it did come with some changes:

- Category slug handling is changed. Historically, Oscar always updated the slug when the name changed. Now a slug is only created if no slug is given, and an existing slug is never overridden. This means that you can freely change the slugs, and a name change will not change the category's URL.
- The `full_name` field has been *removed* and been replaced by a `full_name` property. Accessing that property incurs one database query to fetch the ancestors.
- `Category.get_absolute_url` is now naively cached, as it's more costly to generate the URL than before. But as `ProductCategoryView`, the view returned in `get_absolute_url` only considers the primary key and not the slug, even a stale cache should lead to the correct category page. But if you have altered that logic, please be sure to investigate.

Those changes unfortunately do mean a data migration to update the slugs which *must* be run. Please see the section on migrations below. Please also ensure that, if you load your categories via fixtures, you update them accordingly: remove the `full_name` field and remove the ancestor's slugs from the `slug` field.

Misc

- The `AbstractWeightBased` shipping method now allows zero-weight baskets to have a non-zero shipping cost (#1565). This means that sites that rely on zero-weight baskets having no change will need to introduce a new weight band that covers this edge case.
- The methods **:method:'~oscar.apps.offer.utils.Applicator.apply'** and **:method:'~oscar.apps.offer.utils.Applicatior.get_offers'** changed their arguments to *(basket, user=None, request=None)*. (#1677)

Migrations

Migrations will get picked up automatically for apps you haven't customised. If you have customised any app, please consult the *detailed instructions* on how to handle migrations.

Warning: This release contains a data migration for category slugs. If you have forked it, it is critical you run a copy of that migration when upgrading.

Warning: This release doesn't include any South migrations, as support for Django 1.6 has been dropped.

Note, the catalogue app contains a data migration `0003_data_migration_slugs`. If you have a forked catalogue app, copy this migration into your project so it can be applied (or create a data migration that applies the same transformation).

Dependency changes

- Oscar now requires *django-treebeard* 3.0.

Deprecated features

The following features have been deprecated in this release:

- For backwards compatibility, one can access the `ProductCategoryView` without specifying a category PK in the URL. Oscar itself does not use this any more, and it will be removed with the next version of Oscar.
- `ProductSearchHandler` has been renamed to `SolrProductSearchHandler`. The old name will be removed in the next version of Oscar.

Removal of deprecated features

These methods have been removed:

- `oscar.core.compat.atomic_compat`: Use `django.db.transaction.atomic` instead.
- `oscar.core.loading.import_string`: Use `django.utils.module_loading.import_string` instead.
- `Product.variants`: Use `Product.children`
- `Product.is_top_level`: Use `Product.is_standalone or self.is_parent`
- `Product.is_group`: Use `Product.is_parent`
- `Product.is_variant`: Use `Product.is_child`
- `Product.min_variant_price_incl_tax`: Refactor or use the deprecated `Product.min_child_price_incl_tax`.
- `Product.min_variant_price_excl_tax`: Refactor or use the deprecated `Product.min_child_price_excl_tax`.
- `Strategy.fetch_for_group`: Use `Strategy.fetch_for_parent`.
- `Strategy.select_variant_stockrecords`: Use `Strategy.select_children_stockrecords`.
- `Strategy.group_pricing_policy`: Use `Strategy.parent_pricing_policy`.
- `Strategy.group_availability_policy`: Use `Strategy.parent_availability_policy`.

These instances have been removed:

- `oscar.app.shop`: Use `oscar.app.application` instead.

Oscar 1.1.1 release notes

release 2015-08-05

This is Oscar 1.1.1, a bug fix release.

Bug fixes

- #1802: The glyphicons fonts were missing from the release. This could cause Django's `collectstatic` command to fail.

1.0 release branch

Oscar 1.0 release notes

release 2014-11-07

Welcome to Oscar 1.0! It's been 7 months and some 800 commits since 0.7 and lots of work has gone into 1.0. This release makes quite a few changes, especially around supporting Django 1.7 with its app refactor and new migrations support.

Also, as you might have seen, the repositories for Oscar and most of its extensions have moved to a new Github organisation. This marks a change for Oscar from being a Tangent-sponsored project to a more community-driven one, similar to Django itself. The core team is growing too to accommodate new contributors from outside Tangent. This is an exciting change and we're hopeful that Oscar can continue to grow and prosper. To mark this departure, this release has been renamed from 0.8 (under which we released three beta versions) to 1.0.

Table of contents:

Compatibility

This release adds support for Django 1.7. Per our policy of always supporting two versions of Django, support for Django 1.5 has been dropped.

This release also adds full Python 3.3 and 3.4 support.

If you're using `pysolr` for search, you'll need to upgrade to version 3.2 or later.

What's new in Oscar 1.0?

Explicit differentiation of child, parent and stand-alone products

In some edge cases, it was difficult to determine the 'type' of a product. For example, whether a product is a parent (or "group") product without children or stand-alone product (which never has children). To make that distinction easier, a `structure` field has been introduced on the `AbstractProduct` class. In that process, naming for the three different product structures has been altered to be:

stand-alone product A regular product (like a book)

parent product A product that represents a set of *child* products (eg a T-shirt, where the set is the various color and size permutations). These were previously referred to as "group" products.

child product All child products have a parent product. They're a specific version of the parent. Previously known as product variant.

Some properties and method names have also been updated to the new naming. The old ones will throw a deprecation warning.

Better handling of child products in product dashboard

Together with the changes above, the dashboard experience for child products has been improved. The difference between a parent product and a stand-alone product is hidden from the user; a user can now add and remove child products on any suitable product. When the first child product is added, a stand-alone product becomes a parent product; and vice versa.

In the front-end, the old name of "product variants" has been kept.

Customisation just got easier!

- Oscar's views are now dynamically imported. This means that they can be overridden like most other classes in Oscar; overriding the related Application instance is not necessary any more which simplifies the process of replacing or customising a view.
- A new management command, `oscar_fork_app`, has been introduced to make it easy to fork an Oscar app in order to override one of its classes.

The documentation around *Customising Oscar* has been given an overhaul to incorporate the changes.

Django 1.7 support

Oscar 1.0 comes with support for Django 1.7 out of the box. The app refactor and the new migration framework are both great improvements to Django. Oscar now ships with sets of migrations both for South and the new native migrations framework.

Unfortunately, the changes in Django required a few breaking changes when upgrading Oscar both for users staying on Django 1.6 and for users upgrading to Django 1.7 at the same time. These are detailed in the section for backwards-incompatible changes.

The changes in Django 1.7 meant quite a bit of effort to support both versions of Django, so it's very probable that Django 1.6 support will be removed in the next release of Oscar. Django 1.7 has notable improvements, so with that in mind, we can only recommend upgrading now.

Billing addresses explicitly passed around checkout

The `build_submission` method used in checkout now has a `billing_address` key and the signatures for the `submit` and `handle_order_placement` methods have been extended to include it as a keyword argument. While this change should be backwards compatible, it's worth being aware of the method signature changes in case it affects your checkout implementation.

Dashboard for weight-based shipping methods

There is a new dashboard for weight-based shipping methods. It isn't enabled by default as weight-based shipping methods are themselves not enabled by default. To add it to the dashboard menu, include this snippet in your `OSCAR_DASHBOARD_NAVIGATION` setting:

```
OSCAR_DASHBOARD_NAVIGATION = [
    ...
    {
        'label': _('Shipping charges'),
        'url_name': 'dashboard:shipping-method-list',
    },
    ...
]
```

You'll also need to modify your shipping repository class to return weight-based shipping methods.

US demo site

To help developers building sites for the US, a new example Oscar site has been included in the repo. This customises core Oscar to treat all prices as excluding tax and then calculate and apply taxes once the shipping address is known.

Faceting for category browsing

If Oscar is running with a Solr-powered search backend, the category browsing now shows facets (e.g. filter by price range, or product type). This is implemented via a new `SearchHandler` interface, which will eventually replace the tight coupling between Haystack and Oscar. It therefore paves the way for better support for other search engines.

Reworked shipping app

Several parts of the shipping app have been altered. The most important change is a change to the API of shipping methods to avoid a potential thread safety issue. Any existing Oscar sites with custom shipping methods will need to adjust them to confirm to the new API. The new API and the other changes are detailed below.

See the *backwards incompatible changes* for the shipping app and the *guide to configuring shipping* for more information.

Basket additions clean-up

The forms and views around adding things to your basket have been vigorously reworked. This cleans up some very old code there and ensures variant products are handled in a consistent way.

The changes do require changing the constructor signature of the `AddToBasketForm` - the details are documented in the *Basket app changes*.

Checkout improvements

The checkout process now skips payment if the order total is zero (e.g. when ordering free products or using a voucher). As part of that, checkout views now evaluate *pre-conditions* (as before) and newly introduced *skip conditions*. This should make customising the checkout flow easier.

Out with the old, in with the new

Lots of methods deprecated in the 0.6 release have now been removed. Specifically, the partner "wrapper" functionality is now gone. All price and availability logic now needs to be handled with strategies.

Minor changes

- The `OSCAR_CURRENCY_LOCALE` setting has been removed. The locale is now automatically determined from the current language. This ensures prices are always shown in the correct format when switching languages.
- The login and registration view now redirects staff users to the dashboard after logging in. It also employs flash messages to welcome returning and newly registered users.
- The basket middleware now assigns a `basket_hash` attribute to the `request` instance. This provides a hook for basket caching.
- The tracking pixel now also reports the Oscar version in use. This was forgotten when adding tracking of the Python and Django version in 0.7. Total information collected now is the versions of Django, Python and Oscar.
- The tracking pixel is now served by a server run by the new Oscar organisation, rather than by Tangent.
- The `OSCAR_SLUG_FUNCTION` now accepts both string notation and a callable.
- The default templates now allow the order status to be changed on the dashboard order detail page.
- The forms for the order dashboard views are now loaded dynamically so they can be overridden.
- An `OSCAR_DELETE_IMAGE_FILES` setting has been introduced which makes deleting image files and thumbnails after deleting a model with an `ImageField` optional. It usually is desired behaviour, but can slow down an app when using a remote storage.
- Oscar now ships with a `oscar_populate_countries` management command to populate the country databases. It replaces the `countries.json` fixture. The command relies on the `pycountry` library being installed.
- It is now possible to use product attributes to add a relation to arbitrary model instances. There was some (presumably broken) support for it before, but you should now be able to use product attributes of type `entity` as expected. There's currently no frontend or dashboard support for it, as there is no good default behaviour.
- Payment extensions can now raise a `UserCancelled` payment exception to differentiate between the intended user action and any other errors.
- Oscar has a new dependency, django-tables2. It's a handy library that helps when displaying tabular data, allowing sorting, etc. It also makes it easier to adapt e.g. the product list view in the dashboard to additional fields.
- `jquery-ui-datepicker` has been replaced in the dashboard by bootstrap-datetimepicker. We still ship with `jquery-ui-datepicker` and `JQuery UI` as it's in use in the frontend.
- ... and dozens of bugs fixed!

Backwards incompatible changes in 1.0

Product structure

Generally, backwards compatibility has been preserved. Be aware of the following points though:

- You now need to explicitly set product structure when creating a product; the default is a stand-alone product.
- The `related_name` for child products was altered from `variants` to `children`. A `variants` property has been provided (and will throw a deprecation warning), but if you used the old related name in a query lookup (e.g. `products.filter(variants__title='foo')`, you will have to change it to `children`.
- Template blocks and CSS classes have been renamed.

The following methods and properties have been deprecated:

- `Product.is_parent` - Use `is_group` instead.
- `Product.is_variant` - Use `is_child` instead.
- `Product.is_top_level` - Test for `is_standalone` and/or `is_parent` instead.
- `Strategy.fetch_for_group` - Use `fetch_for_parent` instead.
- `Strategy.group_[pricing|availability]_policy` - Use `parent_[pricing|availability]_policy` instead.
- `Strategy.select_variant_stockrecords` - Use `select_children_stockrecords` instead.

Furthermore, CSS classes and template blocks have been updated. Please follow the following renaming pattern:

- `variant-product` becomes `child-product`
- `product_variants` becomes `child_products`
- `variants` becomes `children`
- `variant` becomes `child`

Product editing

The dashboard improvements for child products meant slight changes to both `ProductCreateUpdateView` and `ProductForm`. Notably `ProductForm` now gets a `parent` kwarg. Please review your customisations for compatibility with the updated code.

Shipping

The shipping method API has been altered to avoid potential thread-safety issues. Prior to v1.0, shipping methods had a `set_basket` method which allowed a basket instance to be assigned. This was really a crutch to allow templates to have easy access to shipping charges (as they could be read straight off the shipping method instance). However, it was also a design problem as shipping methods could be instantiated at compile-time leading to a thread safety issue where multiple threads could assign a basket to the same shipping method instance.

In Oscar 1.0, shipping methods are stateless services that have a method *`calculate()`* that takes a basket and returns a `Price` instance. New *template tags* are provided that allow these shipping charges to be accessed from templates.

This API change does require quite a few changes as both the shipping method and shipping charge now need to be passed around separately:

- Shipping methods no longer have `charge_excl_tax`, `charge_incl_tax` and `is_tax_known` properties.
- The *`OrderCreator`* class now requires the `shipping_charge` to be passed to `place_order`.
- The signature of the *`OrderTotalCalculator`* class has changed to accept `shipping_charge` rather than a `shipping_method` instance.
- The signature of the *`get_order_totals()`* method has changed to accept the `shipping_charge` rather than a `shipping_method` instance.

Another key change is in the shipping repository object. The `get_shipping_methods` method has been split in two to simplify the exercise of providing new shipping methods. The best practice for Oscar 1.0 is to override the `methods` attribute if the same set of shipping methods is available to everyone:

```
from oscar.apps.shipping import repository, methods

class Standard(methods.FixedPrice):
    code = "standard"
    name = "Standard"
    charge_excl_tax = D('10.00')


class Express(methods.FixedPrice):
    code = "express"
    name = "Express"
    charge_excl_tax = D('20.00')

class Repository(repository.Repository):
    methods = [Standard(), Express()]
```

or to override `get_available_shipping_methods` if the available shipping methods if only available conditionally:

```
from oscar.apps.shipping import repository

class Repository(repository.Repository):

    def get_available_shipping_methods(
            self, basket, shipping_addr=None, **kwargs):
        methods = [Standard()]
        if shipping_addr.country.code == 'US':
            # Express only available in the US
            methods.append(Express())
        return methods
```

Note that shipping address should be passed around as instances not classes.

Email address handling

In theory, the local part of an email is case-sensitive. In practice, many users don't know about this and most email servers don't consider the capitalisation. Because of this, Oscar now disregards capitalisation when looking up emails (e.g. when a user logs in). Storing behaviour is unaltered: When a user's email address is stored (e.g. when registering or checking out), the local part is unaltered and the host portion is lowercased.

Warning: Those changes mean you might now have multiple users with email addresses that Oscar considers identical. Please use the new `oscar_find_duplicate_emails` management command to check your database and deal with any conflicts accordingly.

Django 1.7 support

If you have any plans to upgrade to Django 1.7, more changes beyond addressing migrations are necessary:

- You should be aware that Django 1.7 now enforces uniqueness of app labels. Oscar dashboard apps now ship with app configs that set their app label to `{oldname}_dashboard`.
- If you have forked any Oscar apps, you must add app configs to them, and have them inherit from the Oscar one. See the appropriate section in *Forking an app* for an example.
- Double-check that you address migrations as detailed below.
- Django now enforces that no calls happen to the model registry during app startup. This mostly means that you should avoid module-level calls to `get_model`, as that only works with a fully initialised model registry.

Basket line stockrecords

The basket line model got a reference to the stockrecord in Oscar 0.6. The basket middleware since then updated basket lines to have stockrecords if one was missing. If any lines are still missing a stockrecord, we'd expect them to be from from submitted baskets or from old, abandoned baskets. This updating of basket lines has been removed for 1.0 as it incurs additional database queries. Oscar 1.0 now also enforces the stockrecord by making it the `stockrecord` field of basket `Line` model no longer nullable.

There is a migration that makes the appropriate schema change but, before that runs, you may need to clean up your `basket_line` table to ensure that all existing null values are replaced or removed.

Here's a simple script you could run before upgrading which should ensure there are no nulls in your `basket_line` table:

```
from oscar.apps.basket import models
from oscar.apps.partner.strategy import Selector

strategy = Selector().strategy()

lines = models.Line.objects.filter(stockrecord__isnull=True):
for line in lines:
    info = strategy.fetch_for_product(line.product)
    if line.stockrecord:
        line.stockrecord = info.stockrecord
        line.save()
    else:
        line.delete()
```

- The `reload_page_response` method of `OrderDetailView` has been renamed to `reload_page`.

Basket app changes

- The `basket:add` URL now required the primary key of the "base" product to be included. This allows the same form to be used for both GET and POST requests for variant products.

- The `ProductSelectionForm` is no longer used and has been removed.
- The constructor of the `AddToBasketForm` has been adjusted to take the basket and the purchase info tuple as parameters instead of the request instance (c74f57bf and 8ba283e8).

Misc

- The `oscar_calculate_scores` command has been rewritten to use the ORM instead of raw SQL. That exposed a bug in the previous calculations, where purchases got weighed less than any other event. When you upgrade, your total scores will be change. If you rely on the old behaviour, just extend the `Calculator` class and adjust the weights.
- `Order.order_number` now has `unique=True` set. If order numbers are not unique in your database, you need to remedy that before migrating. By default, Oscar creates unique order numbers.
- `Product.score` was just duplicating `ProductRecord.score` and has been removed. Use `Product.stats.score` instead.
- Oscar has child products to model tightly coupled products, and `Product.recommended_products` to model products that are loosely related (e.g. used for upselling). `Product.related_products` was a third option that sat somewhere in between, and which was not well supported. We fear it adds confusion, and in the spirit of keeping Oscar core lean, has been removed. If you're using it, switch to `Product.recommended_products` or just add the field back to your custom Product instance and `ProductForm` when migrating.
- The `basket_form` template tag code has been greatly simplified. Because of that, the syntax needed to change slightly.

  Before: `{% basket_form request product as basket_form single %}`

  After: `{% basket_form request product 'single' as basket_form %}`
- Product attribute validation has been cleaned up. As part of that, the trivial `ProductAttribute.get_validator` and the unused `ProductAttribute.is_value_valid` methods have been removed.
- The `RangeProductFileUpload` model has been moved from the ranges dashboard app to the offers app. The migrations that have been naively drop and re-create the model; any data is lost! This is probably not an issue, as the model is only used while an range upload is in progress. If you need to keep the data, ensure you migrate it across.
- `oscar.core.loading.get_model` now raises a `LookupError` instead of an `ImportError` if a model can't be found. That brings it more in line with what Django does since the app refactor.
- `CommunicationEventType.category` was storing a localised string, which breaks when switching locale. It now uses `choices` to map between the value and a localised string. Unfortunately, if you're using this feature and not running an English locale, you will need to migrate the existing data to the English values.
- Support for the `OSCAR_OFFER_BLACKLIST_PRODUCT` setting has been removed. It was only partially supported: it prevented products from being added to a range, but offers could be applied to the products nonetheless. To prevent an offer being applied to a product, use `is_discountable` or override `get_is_discountable` on your product instances.
- `Category.get_ancestors` used to return a list of ancestors and would default to include itself. For consistency with get_descendants and to avoid having to slice the results in templates, it now returns a queryset of the ancestors; use `Category.get_ancestors_and_self` for the old behaviour.
- Weight based shipping methods used to have an `upper_charge` field which was returned if no weight band matched. That doesn't work very well in practice, and has been removed. Instead, charges from bands are now added together to match the weight of the basket.

- The *OrderCreator* class no longer defaults to free shipping: a shipping method and charge have to be explicitly passed in.
- The Base shipping method class now lives in oscar.apps.shipping.methods.
- The find_by_code method of the shipping Repository class has been removed as it is no longer used.
- The parameters for *oscar.apps.shipping.repository.Repository.get_shipping_methods()* have been re-ordered to reflect which are the most important.
- The legacy ShippingMethod name of the interface of the shipping app has been removed. Inherit from shipping.base.Base for the class instead, and inherit from shipping.abstract_models.AbstractBase for model-based shipping methods.
- oscar.apps.shipping.Scales has been renamed and moved to oscar.apps.shipping.scales.Scale, and is now overridable.
- The models of the shipping app now have abstract base classes, similar to the rest of Oscar.
- The legacy ShippingMethod name of the interface of the shipping app has been removed. Inherit from shipping.base.Base for the class instead, and inherit from shipping.abstract_models.AbstractBase for model-based shipping methods.
- Oscar's models.py files now define __all__, and it's dynamically set to only expose unregistered models (which should be what you want) to the namespace. This is important to keep the namespace clean while doing star imports like from oscar.apps.catalogue.models import *. You will have to check your imports to ensure you're not accidentally relying on e.g. a datetime import that's pulled in via the star import. Any such import errors will cause a loud failure and should be easy to spot and fix.

Migrations

- South is no longer a dependency. This means it won't get installed automatically when you install Oscar. If you are on Django 1.6 and want to use South, you will need to explicitly install it and add it to your requirements.
- Only South >= 1.0 is supported: South 1.0 is a backwards compatible release explicitly released to help with the upgrade path to Django 1.7. Please make sure you update accordingly if you intend to keep using South. Older versions of South will look in the wrong directories and will break with this Oscar release.
- Rename your South migrations directories. To avoid clashes between Django's and South's migrations, you should rename all your South migrations directories (including those of forked Oscar apps) to south_migrations. South 1.0 will check those first before falling back to migrations.
- If you're upgrading to Django 1.7, you will need to follow the instructions to upgrade from South for your own apps. For any forked Oscar apps, you will need to copy Oscar's initial migrations into your emptied migrations directory first, because Oscar's set of migrations depend on each other. You can then create migrations for your changes by calling ./manage.py makemigrations. Django should detect that the database layout already matches the state of migrations; so a call to migrate should fake the migrations.

Warning: The catalogue app has a data migration to determine the product structure. Please double-check it's outcome and make sure to do something similar if you have forked the catalogue app.

Note: The migration numbers below refer to the numbers of the South migrations. Oscar 1.0 ships with a set of new initial migrations for Django's new native migrations framework. They include all the changes detailed below.

Note: Be sure to read the detailed instructions for *handling migrations*.

- Address:
  - `0011` - `AbstractAddress.search_text` turned into a `TextField`.
  - `0012` - `AbstractCountry`: Removed two unused indexes & turns numeric code into `CharField`
- Catalogue:
  - `0021` - Add `unique_together` to `ProductAttributeValue`, `ProductRecommendation` and `ProductCategory`
  - `0022` - Remove `Product.score` field.
  - `0023` - Drop `Product.related_products`.
  - `0024` - Change `ProductAttributeValue.value_text` to a `TextField` and do entity attribute changes and model deletions.
  - `0025` & `0026` - Schema & data migration to determine and save Product structure.
- Offer:
  - `0033` - Use an `AutoSlug` field for `Range` models
  - `0034` - Add moved `RangedProductFileUpload` model.
- Order:
  - `0029` - Add `unique_together` to `PaymentEventQuantity` and `ShippingEventQuantity`
  - `0030` - Set `unique=True` for `Order.order_number`
  - `0031` - `AbstractAddress.search_text` turned into a `TextField`.
- Partner:
  - `0014` - `AbstractAddress.search_text` turned into a `TextField`.
- Promotions:
  - `0006` - Add `unique_together` to `OrderedProduct`
- Ranges dashboard:
  - **`0003` - Drop `RangeProductFileUpload` from `ranges` app. This is** a destructive change!
- Shipping:
  - `0007` - Change `WeightBand.upper_limit` from `FloatField` to `DecimalField`
  - `0008` - Drop `WeightBased.upper_charge` field.

Deprecated features

The following features have been deprecated in this release:

- Many attributes concerning product structure. Please see the product structure changes for details.

Removal of deprecated features

These methods have been removed:

- `oscar.apps.catalogue.abstract_models.AbstractProduct.has_stockrecord`
- `oscar.apps.catalogue.abstract_models.AbstractProduct.stockrecord`
- `oscar.apps.catalogue.abstract_models.AbstractProduct.is_available_to_buy`
- `oscar.apps.catalogue.abstract_models.AbstractProduct.is_purchase_permitted`
- `oscar.apps.catalogue.views.get_product_base_queryset`
- `oscar.apps.partner.abstract_models.AbstractStockRecord.is_available_to_buy`
- `oscar.apps.partner.abstract_models.AbstractStockRecord.is_purchase_permitted`
- `oscar.apps.partner.abstract_models.AbstractStockRecord.availability_code`
- `oscar.apps.partner.abstract_models.AbstractStockRecord.availability`
- `oscar.apps.partner.abstract_models.AbstractStockRecord.max_purchase_quantity`
- `oscar.apps.partner.abstract_models.AbstractStockRecord.dispatch_date`
- `oscar.apps.partner.abstract_models.AbstractStockRecord.lead_time`
- `oscar.apps.partner.abstract_models.AbstractStockRecord.price_incl_tax`
- `oscar.apps.partner.abstract_models.AbstractStockRecord.price_tax`
- `oscar.apps.payment.abstract_models.AbstractBankcard.card_number`

These classes have been removed:

- `oscar.apps.partner.prices.DelegateToStockRecord`
- `oscar.apps.partner.availability.DelegateToStockRecord`
- `oscar.apps.payment.utils.Bankcard`

Known issues

- `models.py` dynamically sets `__all__` to control what models are importable through the star import. A bug in the `models.py` for the `partner` app means you'll have to explicitly import them. More info in #1553.

Oscar 1.0.1 release notes

This is Oscar 1.0.1, a bug fix release.

Bug fixes

- #1553: `from oscar.apps.partner.models import *` could lead to the wrong models being imported.
- #1556: Dashboard order table headers shifted

- #1557: Fixed an issue where Oscar wrongly used Django's `is_safe_url`. Hence some redirects might not have worked as expected. This change unfortunately meant updating the notation of `oscar.core.utils.safe_referrer()` and `oscar.core.utils.redirect_to_referrer()` to accept the request instead of request.META.
- #1577: The billing address was not being correctly passed through to the *place_order* method.
- #1592: `Product.min_child_price_[excl|incl]_tax` were broken and failing loudly. They are not recommended any more, but to ensure backwards-compatibility, they have been fixed.

Oscar 1.0.2 release notes

This is Oscar 1.0.2, a bug fix release.

Bug fixes

- #1562: Correctly handle the update_fields kwarg on `Category.save()`
- #1672: `Order.shipping_status` was not guaranteed to return the correct status.

0.7 release branch

Oscar 0.7 release notes

release 2014-04-29

Welcome to Oscar 0.7!

These release notes cover the *new features* as well as *backwards incompatible changes* that you'll want to be aware of when upgrading from Oscar 0.6 or earlier.

If you encounter any undocumented issues, please let us know on the mailing list.

Table of contents:

Overview

Oscar 0.7 is largely a maintenance release, fixing minor issues, merging long-standing pull requests and other housekeeping.

As part of the clean-up, we have removed a few unused models and model fields, as well as removing `null=True` from a load of `CharFields` - so please read the release notes carefully when upgrading as some schema migrations may need some care.

Further, ensure you test your checkout implementation carefully after upgrading as the core Oscar checkout view classes have been reorganised slightly. Any upgrading work should be minor but be diligent.

Compatibility

Oscar 0.7 has experimental support for Python 3.

Support for Django 1.4 has been dropped, and support for Django 1.6 is now considered stable.

What's new in Oscar 0.7?

Search improvements

Several improvements have been made to Oscar's default search functionality:

- Search results can be now be sorted.
- If your search backend supports it, spelling suggestions will be shown if the original search term doesn't lead to any results.
- Only products are returned by the core search view. Other content types in your search index are filtered out (#370).

Extended signals

Oscar's signals have been improved and consolidated, making it easier to hook into user journeys and extract analytics information.

Changes to existing signals include:

- The *basket_addition* signal now passes the `request` as an additional kwarg.
- The *user_registered* signal now passes the `request` as an additional kwarg.

New signals:

- A *start_checkout* signal is now raised when the customer begins the checkout process.

See the *signals docs* for more details.

Checkout reorganisation

The checkout classes have been reworked to clean-up how pre-conditions are enforced. Each view class now has a `pre_conditions` attribute which is an iterable of method names (as strings). Each method is run within the `dispatch` method of the view and will redirect the customer back to the appropriate view if the check fails.

This change makes pre-conditions easier to customise and simplifies the core checkout views. Consequently, the following methods are no longer required and have been removed:

- `PaymentDetails.get_error_response`
- `PaymentDetails.can_basket_be_submitted`

Further, the `PaymentDetailsView` has been re-organised for extensibility. For instance, several new methods have been introduced to allow fine-grained overriding of functionality:

- *`handle_payment_details_submission()`* - This is responsible for validating any forms submitted from the payment URL
- *`handle_place_order_submission()`* - This is responsible for placing an order after a submission from the preview URL.

- *render_payment_details()* - Render the payment details template.

The implementation of *submit()* has been improved to handle payment errors in a more customer friendly way. If an exception is raised during payment, the payment details page is now loaded with the original forms passed to the template (so form validation errors can be displayed).

Finally, the `billing_address` kwarg to `submit`() has been removed. If you want to pass a billing address to be saved against the order, then pass it as part of the `order_kwargs` option.

Minor changes

- Oscar's LESS files now use Bootstrap 2.3.2 (Oscar 0.6 uses 2.1.1).
- The product model now has a `num_approved_reviews` property to avoid unnecessary SQL queries when rendering templates (#1299)
- Customers can delete their profiles from within their account section.
- Customers are prevented from using short or common passwords when changing their password in their account (#1202)
- *permissions_map* now supports more than two lists to evaluate permissions.
- Formset handling in `ProductCreateUpdateView` has been simplified and now easily allows adding further formsets.
- Increased required version of Django Haystack to 2.1
- The dashboard's Bootstrap and the Bootstrap JS has been bumped to 2.3.2, the latest release of version 2.
- The dashboard's category handling now has the ability to directly create child categories.
- Oscar's error messages now have their own CSS class, `error-block` (ef3ccf08a7).
- It is now possible to disable the redirect that happens when a product or category's slug changed and an old URL is used (b920f8ba).
- `BankCardNumberField` now allows specifying accepted card types (32b7249).
- Several slug fields have been turned into the newly introduced `AutoSlugField` to ensure that generated slugs are unique.
- Widget initialisation can now be prevented with adding the `no-widget-init` class. Issues around widget initialisation in the dashboard promotions have been resolved.
- The access function used to determine dashboard's menu entries' visibility is now settable via *OSCAR_DASHBOARD_DEFAULT_ACCESS_FUNCTION*.
- Vouchers start and end times are now datetimes instead of dates; allowing "lunch-time deals" etc.
- Product classes can now be added from the dashboard. Editing options and attributes is not yet supported though.
- Experimental support for having a language prefix in the URL has been added, and enabled for the sandbox. This can be achieved by using Django's i18n_patterns function in your `urls.py`. for the sandbox. See `sites/sandbox/urls.py` for an example.
- A basic example for a multi-language sitemap has been added to the sandbox.
- Reasoning about e.g. when it is feasible to drop Python 2.6 or Django 1.5 support is hard without reliable data, hence the tracker pixel has been extended to submit the Python and Django version in use. Tracking is still easily disabled by setting `OSCAR_TRACKING` to `False`.

Bugfixes

- Addresses in non-shipping countries can no longer be selected as default shipping address anymore (be04d46639).
- Suspended and consumed offers are no longer returned by the "active" offer manager. (#1228).
- Products can now be removed from categories (#1289).

Backwards incompatible changes in 0.7

Warning: Fields and models have been removed from Oscar. If you used them, you must ensure you create/extend the affected models appropriately.

- Oscar has dropped support for Django 1.4. However, if Oscar continues to support the `AUTH_PROFILE_MODULE` setting so sites that use separate profile models aren't forced to convert to a single user model in order to use Oscar 0.7.
- `AbstractProduct.status` was an unused `CharField` provided for convenience as it's a commonly required field. But a different field type was often required, and as changing it is much harder than adding a field with the desired type, the field has been removed.
- `Contributor`, `ContributorRole`, the through-model `ProductContributor` and their abstract versions have been removed as they were unused and too specific to the domain of book shops.
- `ProductCategory.is_canonical` was an unused `BooleanField` and has been removed.
- `Order.basket_id` was a `PositiveIntegerField` containing the primary key of the associated basket. It's been refactored to be a nullable `ForeignKey` and is now called "basket".
- #1123 - The URL structure of `ProductCreateRedirectView` has been changed to use the product class' slug instead of the primary key. It's necessary to update URLs pointing to that view.
- `ProductListView` has been removed as it wasn't needed any more after the search improvements. The old URL route still works.
- Accessing categories by just slug instead of primary key and slug had been unofficially deprecated for 0.6, and is removed now.
- #1251 - Form related templates have been refactored. If you've modified them, your templates might need updating.
- `django.conf.urls.i18n` has been removed from Oscar's default URLs. This is because to get `i18n_patterns` working for Oscar, it needs to be defined outside of the scope of it. If you use i18n, you need to explicitly add the following line to your main `urls.py`:

```
(r'^i18n/', include('django.conf.urls.i18n')),
```

- jScrollPane, which was used to style the dashboard's scroll bars, has been removed.
- The methods `get_error_response` and `can_basket_be_submitted` have been removed from the `PaymentDetailsView` view class in checkout

Removal of features deprecated in 0.6

- Django 1.4 support has been removed.

- In *OrderPlacementMixin*, the following methods have been removed:
  - `create_shipping_address_from_form_fields` - This is removed as checkout now requires an unsaved shipping address instance to be passed in (rather than having it created implicitly).
  - `create_user_address` - This is replaced by `oscar.apps.checkout.mixin.OrderPlacementMixin.update_address_book()`.
  - `create_shipping_address_from_user_address`
- The `oscar.apps.checkout.session.CheckoutSessionData.shipping_method()` has been removed. Instead *oscar.apps.checkout.session.CheckoutSessionMixin.get_shipping_address()* provides the same functionality.

Migrations

Warning: The reviews app has not been under migration control so far. Please ensure you follow South's guidelines on how to convert an app. Essentially, you will have to run: `$ ./manage.py migrate reviews 0001 --fake`

Warning: A lot of Oscar apps have data migrations for CharFields before `null=True` is removed in the following schema migration. If you have extended such an app and use your own migrations, then you will need to first convert affected `None`'s to `''` yourself; see the data migrations for our approach.

Note: Be sure to read the detailed instructions for *handling migrations*.

- Address:
  - `0008` - Forgotten migration for `UserAddress.phone_number`
  - `0009` & `0010` - Data and schema migration for removing `null=True` on `CharFields`
- Catalogue:
  - `0014` - Drops unused `ProductCategory.is_canonical` field.
  - `0015` - Turns a product's UPC field into a *oscar.models.fields.NullCharField*
  - `0016` - `AutoSlugField` for `AbstractProductClass` and `AbstractOption`
  - `0017` - Removes `Product.status`, `Contributor`, `ContributorRole` and `ProductContributor`
  - `0018` - Set `on_delete=models.PROTECT` on `Product.product_class`
  - `0019` & `0020` - Data and schema migration for removing `null=True` on `CharFields`
- Customer:
  - `0006` - `AutoSlugField` and `unique=True` for `AbstractCommunicationEventType`
  - `0007` & `0008` - Data and schema migration for removing `null=True` on `CharFields`
  - `0009` - Migration caused by `CommunicationEventType.code` separator change
- Offer:

  - 0029 - AutoSlugField for ConditionalOffer
  - 0030 & 0031 - Data and schema migration for removing null=True on CharFields
  - 0032 - Changing proxy_class fields to NullCharField
- Order:
  - 0025 - AutoSlugField for AbstractPaymentEventType and AbstractShippingEventType‘‘
  - 0026 - Allow null=True and blank=True for Line.partner_name
  - 0027 & 0028 - Data and schema migration for removing null=True on CharFields
- Partner:
  - 0011 - AutoSlugField for AbstractPartner
  - 0012 & 0013 - Data and schema migration for removing null=True on CharFields
- Payment:
  - 0003 - AutoSlugField and unique=True for AbstractSourceType
- Promotions:
  - 0004 & 0005 - Data and schema migration for removing null=True on CharFields
- Shipping:
  - 0006 - AutoSlugField for ShippingMethod
- Reviews:
  - 0001 - Initial migration for reviews application. Make sure to follow South's guidelines on how to convert an app.
  - 0002 & 0003 - Data and schema migration for removing null=True on CharFields
- Voucher:
  - 0002 and 0003 - Convert [start|end]_date to [start|end]_datetime (includes data migration).

Oscar 0.7.1 release notes

This is Oscar 0.7.1, a nano-release to squash one gremlin in 0.7 that affects django-oscar-paypal.

Bug fixes

This release makes a change to the checkout session mixin which allows a basket to be explicitly specified by subclasses of the checkout PaymentDetails view class. This is required when a different basket to request.basket is intended to be used in a preview (this is what django-oscar-paypal needs to do).

Oscar 0.7.2 release notes

This is Oscar 0.7.2, a minor security release. If you rely on the permissions_required decorator or the Application.permissions_map and Application.default_permissions syntax, you must upgrade.

Bug fixes

- The `permissions_required` decorator now handles both methods and properties on the User model. Previously, it wasn't supported, but a docstring showed `is_anonymous` as an example, which is a method.
- It fixes a syntax error in `basket.views.BasketView` when rendering an error message. Previously, trying to save an item for later while not being logged in would cause an Internal Server Error.

Oscar 0.7.3 release notes

This is Oscar 0.7.3, a tiny release to relax the dependency restrictions for South and django-compressor:

- South 1.0 can now be used with Oscar 0.7.3. Previously it was capped at 0.9.
- django-compressor 1.4 can now be used with Oscar 0.7.3 when running Python 2.7 (Previously 1.3 was specified for Python 2.7, while 1.4a was specified for Python 3).

0.6 release branch

Oscar 0.6 release notes

release 2014-01-08

It took a while but it's finally here: welcome to Oscar 0.6!

These release notes cover the *new features* as well as *backwards incompatible changes* that you'll want to be aware of when upgrading from Oscar 0.5 or earlier. This release contains some major changes to core APIs which means many old APIs are scheduled to be dropped - see the *deprecation plan* to avoid any nasty surprises.

When upgrading your Oscar site, make sure you study both the *backwards incompatible changes* and the *deprecated features*. If you encounter any undocumented issues, please let us know on the mailing list. Table of contents:

Overview

The biggest change in Oscar 0.6 is the reworking of *pricing and availability*, which builds on top of the change to allow *multiple stockrecords per product*. The change is largely backwards compatible with the old system of "partner wrappers" but it is recommended to upgrade to the new system. Support for partner wrappers will be removed for Oscar 0.7.

Oscar 0.6 also introduces better support for marketplace-like functionality with the so-called permission-based dashboard. It is now possible to give non-staff users access to a subset of the dashboard's views (products and orders) by setting the new `dashboard_access` permission.

Oscar now supports Django 1.5 and its custom user model. This has been only tested in the context of starting a new Oscar project with a custom model. Switching from a separate "profile" model to the new system is not recommended at this point.

Oscar also supports Django 1.6 although this is considered experimental at this stage. It's possible there are still some incompatibilities that haven't been teased out just yet.

Other notable new features include:

- A feature-rich *demo site* that illustrates how Oscar can be customised. It uses several of Oscar's many extensions such as django-oscar-paypal, django-oscar-datacash and django-oscar-stores. It is intended as a reference site for Oscar.
- *Partners can now have addresses*.
- *Customer wishlists*. Customers can how add products to wishlists and manage them within their account section.
- *New helper methods* in the `EventHandler` class for order processing.
- *Reworked search app* with support for easy faceting.

Also, to help justify Tangent's sponsorship of Oscar, a simple *tracking mechanism* has been introduced to keep track of which sites use Oscar.

What's new in Oscar 0.6?

Multiple stockrecords per product

Products can now have multiple stockrecords rather than just one. This is a key structural change that paves the way for many advanced features.

If a product can be fulfilled by multiple partners, a different stockrecord can be created for each partner. This is a common requirement for large-scale e-commerce sites selling millions of products that use many different fulfillment partners.

It also allows better support for international sites as stockrecords can be created for partners in different countries, who sell in different currencies.

See the *documentation on pricing and availability* for more details.

> **Warning:** This changes means several APIs are *deprecated* as they assume there is only one stockrecord per product.

Pricing and availability

When products can have many stockrecords, a process needs to be in place to choose which one is selected for a given customer and product. To handle this, a new "strategy" class has been introduced, responsible for selecting the appropriate stockrecord for a given customer and product.

This change also paved the way for reworking how prices, taxes and availability are handled. Instead of using *"partner wrappers"*, the strategy class is responsible for returning availability details and prices for a particular product. New classes known as pricing and availability policies are used to cleanly encapsulate this information.

These changes allow Oscar to dynamically determine prices, partner and availability for a given customer and product. This enables several advanced features such as:

- Fulfilling a product from the partner that offers the best margin.
- Fulfilling a product from the partner geographically closest to the customer.
- Automatically switching to a new partner when when stock runs out.

- Supporting transactions in multiple currencies on the same site.
- Supporting different tax treatments on the same site (eg UK VAT and US sales tax)
- Having different pricing and availability policies for different customers.

More generally, it provides a structure for customising how pricing, availability work on a per-customer basis. This gives a great deal of flexibility.

See the guide to *prices and availability* for more information.

Permission-based dashboard

Three changes were necessary to better support marketplace scenarios within Oscar:

- Oscar's core `Application` class now supports specifying permissions on a per-view basis. This is done via a new default decorator. Legacy behaviour is unchanged.
- The dashboard's menus are now built dynamically. If the current user does not have access to some views in *OSCAR_DASHBOARD_NAVIGATION*, they will be omitted in the menu returned by `oscar.apps.dashboard.nav.create_menu()`.
- The index, catalogue and order dashboard views have been modified to allow access to non-staff users. See *the dashboard documentation* for details.
- The relation `oscar.apps.partner.abstract_models.AbstractPartner.users` was not used by core Oscar prior 0.6. It is now used to model access for the permission-based dashboard.

Payment models have abstract versions

The models within the *payment app* have been split into abstract and concrete versions. This brings them inline with other Oscar apps and allows them to be customised in the normal way.

Wishlists

Wishlist functionality has finally landed. Signed in customers are now able to create multiple named wishlists and add products to them. There is a new section in the customer's account where wishlists can be managed.

See the *wishlist documentation* for more details.

Partner dashboard & addresses

Partners can now have addresses. These are useful for US sales tax where tax calculations need to know the origin of a product being shipped.

A dashboard to handle partners, their users and addresses has been added.

Checkout

The `PaymentDetailsView` checkout view has been restructured for flexibility. There is a new `build_submission()` method which is responsible for building a dict of all data for passing to the `submit` method. This includes the shipping address and shipping method which were previously loaded indirectly within the `submit` method.

Fig. 1.1: *The add-to-wishlist button.*

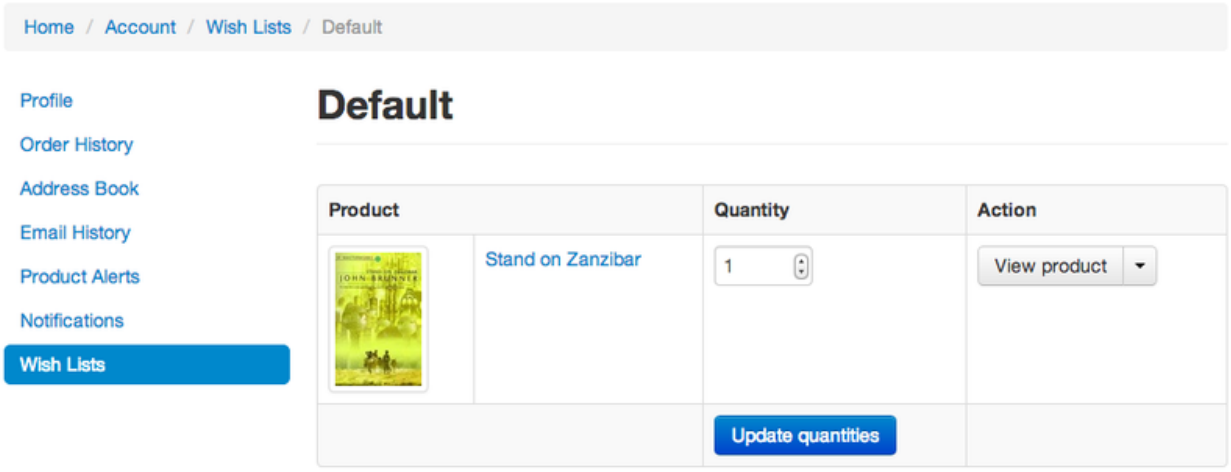

Fig. 1.2: *Editing a wishlist*

Warning: While not major, the changes to checkout are backwards incompatible. See the *backwards compatibility notes* for more details.

Demo site

Oscar now ships with a demo site along side the sandbox site. While the sandbox is a minimal Django project that uses Oscar with all its defaults, the demo site is a more realistic example of an Oscar project. It has a custom skin and makes many alterations to the default Oscar behaviour.

It's features include:

- A range of different product types: books, downloads, clothing
- PayPal Express integration using django-oscar-paypal
- Datacash integration using django-oscar-datacash

See the *sandbox and demo site documentation* for more details. A publicly accessible version of the demo site is available at http://demo.oscarcommerce.com.

Django 1.5, 1.6 and custom user model support

Oscar now supports Django 1.5 and, experimentally, 1.6.

Specifically, Oscar supports custom user models, the headline new feature in Django 1.5. These can be used standalone or with a one-to-one profile model: Oscar's account forms inspect the model fields to dynamically pick up the fields for editing and display.

There are some restrictions on what fields a custom user model must have. For instance, Oscar's default auth backend requires the user model to have an email and password field. New Oscar projects are encouraged to use the provided abstract user model as the base for their users.

Support for Django 1.6 is considered experimental at the moment as there hasn't been time to run thorough tests for all possible incompatibilities.

Further reading:

- *How to use a custom user model*.

Accounts

The views and templates of the accounts section have been reworked to be clearer and easier to extend. There is no longer a generic frontpage for the accounts section - instead, each subsection has its own page. The navigation has also moved to the left-hand side.

Bootstrap-WYSIHTML5 replaced by TinyMCE

TinyMCE 4.0 is now used in the dashboard for all textareas with class `wysiwyg`. This has better browser support and is easier to customise than bootstrap-wysihtml5 (which has now been removed).

It is easy to configure or replace with the HTML editor of your choice.

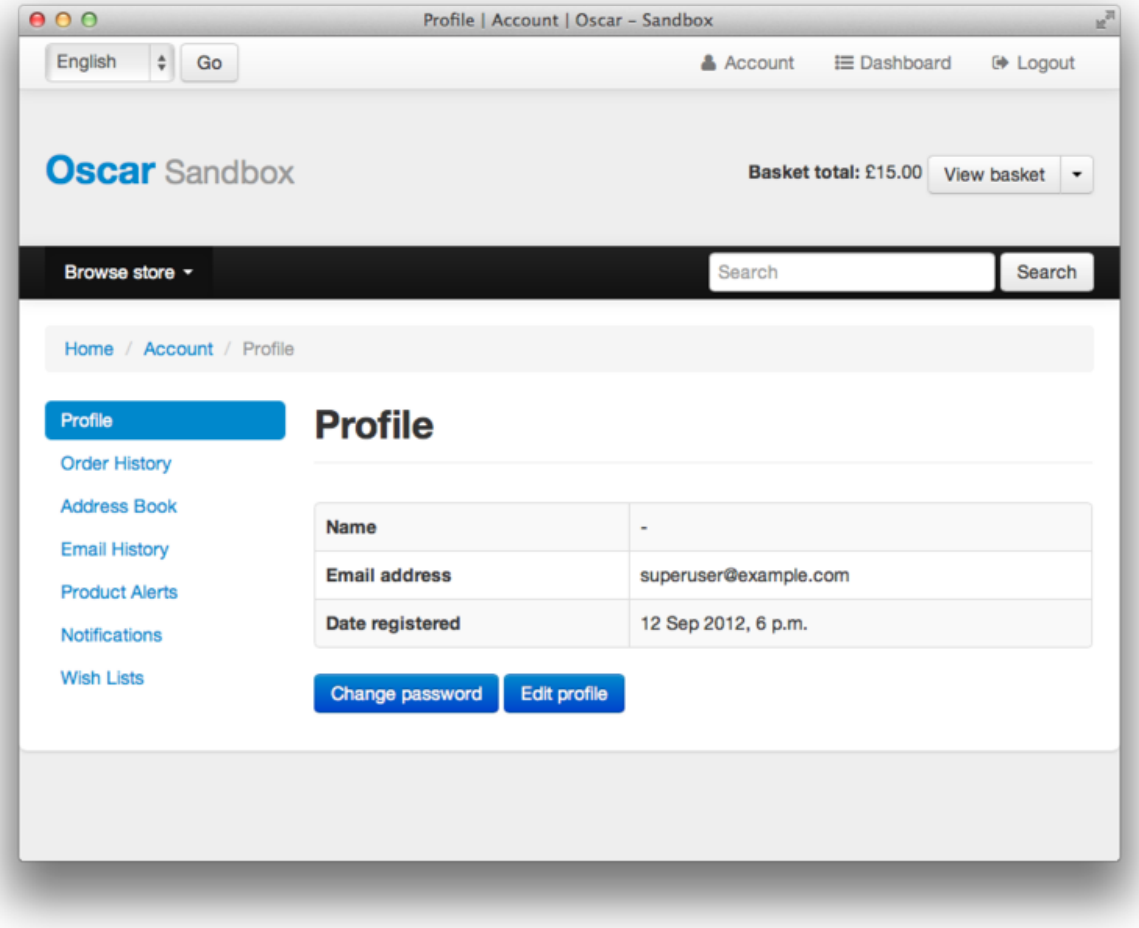

Fig. 1.3: *The new-look account section with navigation on the left-hand side.*

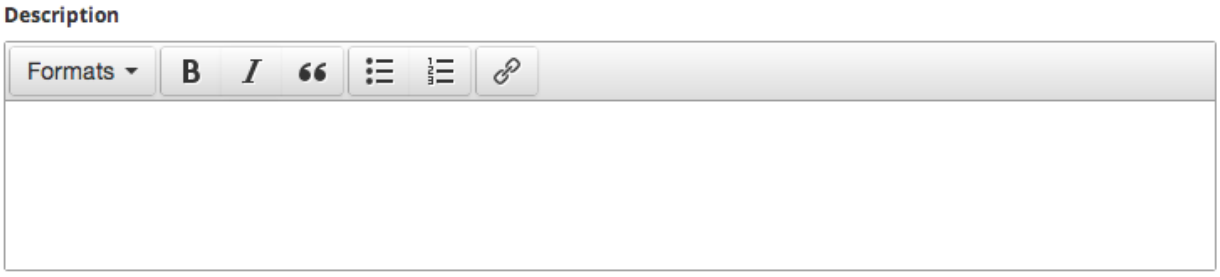

Fig. 1.4: *Textarea with class ‘‘wysiwyg‘‘ now use TinyMCE.*

Improved address fields

The postcode and phone number fields have been improved.

- The postcode field is now validated in the model's `clean()` method to ensure it is valid for the selected country.
- The phone number field now uses a specialist `PhoneNumberField` field class which validates and cleans the phone number.

Better bankcard handling

In 0.5, there were two classes that representing a bankcard. These have been merged - the new version is *`AbstractBankcard`*.

An instance of this model is returned by the `bankcard` property.

Customer-facing range pages

Ranges can now be flagged as public which means they get a customer-facing detail page which shows a range description and allows its products to be browsed.

In the dashboard, the display order of the range's products can be controlled.

To this end, the core *`Range`* model has been extended with a HTML description field.

Reworked search app

Oscar's search app has been reviewed and simplified. The main view class (now *`FacetedSearchView`*) has been reworked to provide better support for faceting, which can be easily specified using the *OSCAR_SEARCH_FACETS* setting.

The `SuggestionsView` has been removed as it wasn't being used. A later version of Oscar will include a replacement.

See the *search app documentation* for more details.

Order processing changes

The core *`EventHandler`* class has been extended.

- The `handle_shipping_event` method now validates a proposed shipping event before saving it.
- The `handle_payment_event` method now validates a proposed payment event before saving it.

See the *`EventHandler`* for the available methods.

Tracking Oscar sites

Oscar's dashboard now serves a single pixel image from one of Tangent's servers. This is included to gather information on which sites use Oscar, which is an important metric for Tangent, who sponsor Oscar's development.

It can easily be disabled by setting `OSCAR_TRACKING=False`. If you do opt out, please email the mailing list with any production Oscar sites you are running. This will help to ensure investment in Oscar's future.

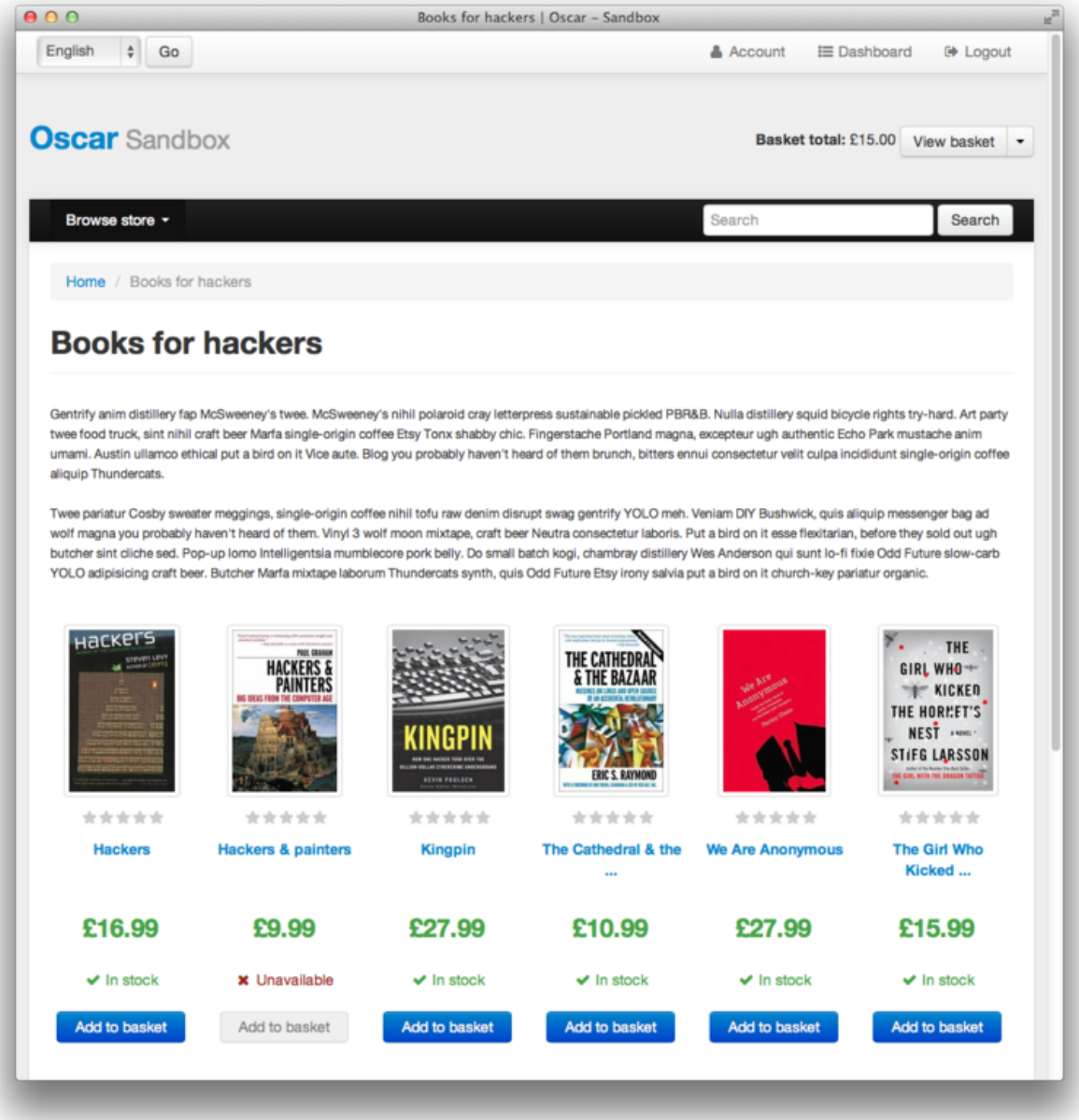

Fig. 1.5: *A customer-facing range page*

Minor changes

- detox is a new dependency, which allows running *tox* tests in parallel.
- `OSCAR_ALLOW_ANON_REVIEWS` has been a documented setting since Oscar 0.4. But there's never been any code to support this, which has been rectified with this release. Things should now work as expected.
- Oscar uses a cookie to display recently displayed products. This cookie never expired and wasn't a `HttpOnly` cookie. It is now a `HttpOnly` cookie and expires after 7 days. Additionally, two settings have been introduced to configure it analogues to the basket cookies: `OSCAR_RECENTLY_VIEWED_COOKIE_LIFETIME` and `OSCAR_RECENTLY_VIEWED_COOKIE_NAME`.

Backwards incompatible changes in 0.6

There were quite a few backwards incompatible changes in Oscar 0.6. There shouldn't be quite as many in future Oscar releases as we approach 1.0.

Additional apps

Four new apps are required in your `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    ...
    'oscar.apps.wishlists',
    'oscar.apps.dashboard.pages',
    'oscar.apps.dashboard.partners',
    'oscar.apps.dashboard.reviews',
    ...
)
```

If you are using the `get_core_apps` helper function, then these new apps will be added automatically. The new wishlists app contains database migrations, so you will need to run the `migrate` management command.

Checkout app

Several changes have been made to the checkout in the name of simplification and making things easier to customise.

The `PaymentDetailsView` has been adjusted to explicitly pass variables around rather than relying on methods that load them on demand. This makes customisation easier and everything more explicit (a good thing).

- The `submit` method in *`PaymentDetailsView`* has a new signature. It now accepts the user, shipping address, shipping method and order total as required parameters The intention is that the `build_submission` methods returns a dict of kwargs for `submit` so that it can be called like:

  ```
  submission = self.build_submission()
  return self.submit(**submission)
  ```

  If your payment or order submission process requires additional parameters (eg a bankcard instance), override the `build_submission` method to provide them. The dict returned from the new `build_submission` method is also passed to the template.

- The `handle_payment` method in *`PaymentDetailsView`* now accepts a *`Price`* instance instead of a Decimal for the order total.

- The `handle_order_placement` method in *`OrderPlacementMixin`* now accepts the user, shipping address and shipping method in a different order consistent with the `submit` method from `PaymentDetailsView`.
- The `place_order` method in *`OrderPlacementMixin`* has a new signature. The parameters have been reordered and the shipping address, shipping method and billing address must be passed in explicitly (as unsaved instances).
- The `create_shipping_address` method in *`OrderPlacementMixin`* has changed signature. Instead of being passed a basket, it is now passed the user and an unsaved shipping address instance.
- The `create_billing_address` method in *`OrderPlacementMixin`* has changed signature. It is now passed an unsaved billing address instance as well as a shipping address instance.
- The `get_shipping_method` (from *`CheckoutSessionMixin`*) no longer defaults to returning free shipping if no shipping method can be looked up.
- The *`OrderTotalCalculator`* now returns a *`Price`* instance from a new `calculate` method. The old methods `order_total_incl_tax` and `order_total_excl_tax` have been removed.

Other changes:

- The checkout gateway page has a new option "Register and continue" which allows a customer to register before checking out. As part of this change, the option value `new` in `GatewayForm` has changed to `guest` as `new` option is used for this feature.
- The checkout decorators `basket_required` and `prev_steps_must_be_complete` have been removed as they were no longer used.

Shipping app changes

The default implementation of the *`Repository`* class has been adjusted to avoid thread-safety issues. If you define your own shipping `Repository` class, ensure that your shipping methods are instantiated per-request and not at compile time.

For example, avoid this:

```
from oscar.apps.shipping import repository

class Repository(repository.Repository)
    # Don't instantiate at compile time!
    methods = [SomeMethod(), AnotherMethod()]
```

Instead, instantiate the methods within `get_shipping_methods`:

```
from oscar.apps.shipping import repository

class Repository(repository.Repository)
    # Note, methods are not instantiated.  The get_shipping_methods
    # method will instantiate them.
    methods = [SomeMethod, AnotherMethod]
```

> **Warning:** Beware of shipping methods that are configured via constructor parameters, like *`FixedPrice`*. If you are using methods that work this way, ensure you instantiate them at runtime.
>
> Shipping methods will be reworked for Oscar 0.7 to avoid these issues.

Address model changes

- The `UserAddress.salutation` and `UserAddress.name` methods are now properties.
- The `Country` model's `is_highlighted` column has been renamed to `display_order` and is now an integer field to allow fine-grained country selection.

Basket app changes

Several properties of the basket `Line` model have been renamed:

- `Line.line_price_excl_tax_and_discounts` has been renamed to `Line.line_price_excl_tax_incl_discounts`.
- `Line.line_price_incl_tax_and_discounts` has been renamed to `Line.line_price_incl_tax_incl_discounts`.

The `basket_form()` templatetag has been altered to take the `request` as the first parameter, not `request.basket`.

Catalogue app changes

3 properties have been removed from *oscar.apps.catalogue.abstract_models.AbstractProductImage* as they were unused: `resized_image_url`, `fullsize_url` and `thumbnail_url`. Thumbnailing is instead achieved in templates with Sorl.

- The function `add_category_from_breadcrumbs` was not used and has been removed.
- Alternative product class templates now use `slug` field instead of `name.lower()` to determine their filename. If you have templates for specific product classes, please update your filenames accordingly

Customer app changes

The *oscar.apps.customer.forms.EmailAuthenticationForm* form now needs to be instantated with a host name so prevent redirects to external sites.

Offer app changes

The ManyToManyField `included_product` of the *Range* model was changed to use `through` relationship:

- Use `Range.add_product(product)` instead of `Range.included_product.add(product)`.
- Use `Range.remove_product(product)` instead of `Range.included_product.remove(product)`.

When adding a product into a range, position in the range can be specified with `display_order` parameter: `Range.add_product(product, display_order=3)`

Payment app changes

The balance method on the *AbstractSource* model is now a property, not a method.

Reviews app changes

The two product review forms, `SignedInUserProductReviewForm` and `AnonymousUserProductReviewForm`, have been replaced by a new `oscar.apps.catalogue.reviews.forms.ProductReviewForm`.

Search app changes

Some of the names have been simplified.

- The `MultiFacetedSearchView` and `SuggestionsView` view classes have been removed. The `MultiFacetedSeachView` class is replaced by `FacetedSearchView`.
- The `MultiFacetedSearchForm` has been removed in favour of `SearchForm`.

Loading baskets

Now that products can have multiple stockrecords, several changes have been made to baskets to allow the appropriate stockrecord to be tracked for each basket line. The basket line model has a new field that links to the selected stockrecord and the basket itself requires an instance of the strategy class so that prices can be calculated for each line. Hence, if you loading baskets and manipulating baskets directly, you need to assign a strategy class in order for prices to calculate correctly:

```
from oscar.apps.basket import models

basket = models.Basket.objects.get(id=1)
basket.strategy = request.strategy
```

Without an assigned strategy class, a basket will raise a `RuntimeError` when attempting to calculate totals.

Renamed templates

Some templates have been renamed for greater consistency. If you are overriding these templates, ensure you rename your corresponding project templates.

Many of the profile templates have been reorganised:

- `customer/address_list.html` is renamed to `customer/address/address_list.html`
- `customer/address_form.html` is renamed to `customer/address/address_form.html`
- `customer/address_delete.html` is renamed to `customer/address/address_delete.html`
- `customer/email.html` is renamed to `customer/email/email_detail.html`
- `customer/email_list.html` is renamed to `customer/email/email_list.html`
- `customer/order.html` is renamed to `customer/order/order_detail.html`
- `customer/order_list.html` is renamed to `customer/order/order_list.html`
- `customer/profile.html` is renamed to `customer/profile/profile.html`
- `customer/profile_form.html` is renamed to `customer/profile/profile_form.html`
- `customer/change_password_form.html` is renamed to `customer/profile/change_password_form.html`

- `partials/nav_profile.html` has been removed.

Template block changes

- The template `dashboard/orders/order_detail.html` has been reorganised. The `tab_transactions` block has been renamed to `payment_transactions`.
- In `checkout/checkout.html`, the `checkout-nav` block has been renamed `checkout_nav`.

Changes to Partner permissions

The following permissions on the *`AbstractPartner`* model were not used in Oscar and have been removed to avoid confusion with the newly introduced permission-based dashboard:

- `can_edit_stock_records`
- `can_view_stock_records`
- `can_edit_product_range`
- `can_view_product_range`
- `can_edit_order_lines`
- `can_view_order_lines`

The permission-based dashboard introduced a new permission:

- `dashboard_access`

Migrations

There are rather a lot of new migrations in Oscar 0.6. They are all detailed below.

If you are upgrading and your project overrides one of these apps with new migrations, then ensure you pick up the schema changes in a new migration within your app. You can generally do this using `manage.py schemamigration $APP --auto` but check the corresponding core migration to ensure there aren't any subtleties that are being overlooked.

Some of these migrations rename fields for consistency, while others ensure `CharField` fields are not nullable.

- Address:
  - `0003`: A new field `display_order` is added to the `Country` model. This is the first of 3 migrations that replace the boolean `is_highlighted` field with an integer field that allows fine-grained control of the order of countries in dropdowns.
  - `0004`: A data migration to ensure highlighted countries have a display order of 1.
  - `0005`: Remove the `is_highlighted` field from the `Country` model as it is no longer necessary.
  - `0006`: Add a uniqueness constraint across `user_id` and `hash` for the `UserAddress` model to prevent duplicate addresses.
  - `0007`: Use a custom field for address postcodes.
- Basket:
  - `0004`: Add `stockrecord` field to the `Line` model to track which stockrecord has been selected to fulfill a particular line.

  - 0005: Add price_currency field to the Line model.
- Catalogue:
  - 0011: Larger max_length on FileFields and ImageFields
  - 0012: Use NullBooleanField for the value_boolean field of the ProductAttributeValue model.
  - 0013: Add value_file and value_image fields to the ProductAttributeValue model to support file and image attributes.
- Customer:
  - 0005: Don't allow sms_template field of CommunicationEventType model to be nullable.
- Dashboard:
  - 0002: Don't allow error_message field of RangeProductFileUpload model to be nullable.
- Offer app:
  - 0020: Data migration to set null offer descriptions to empty string.
  - 0021: Don't allow null offer descriptions or benefit types.
  - 0022: Add a slug field to the *Range* model.
  - 0023: A data migration to populate the new range slug field.
  - 0024: Add a is_public field to the *Range* model.
  - 0025: Add a description field to the *Range* model.
  - 0026: Add a applies_to_tax_exclusive_price field to ConditionalOffer model to try and handle offers that apply in bothe the US and UK (this field is later removed).
  - 0027: Create a joining table for the new M2M relationship between ranges and products.
  - 0028: Remove applies_to_tax_exclusive_price field.
- Order app:
  - 0010: Allow postcodes for shipping- and billing addresses to be nullable.
  - 0011: Rename date field on CommunicationEvent, ShippingEvent and PaymentEvent models to be date_created.
  - 0012: Add reference field to PaymentEvent model.
  - 0013: Add a foreign key to ShippingEvent from PaymentEvent model.
  - 0014: Change postcode field on ShippingAddress and BillingAddress models to use UppercaseCharField field.
  - 0015: Remove is_required and sequence_number fields from ShippingEventType and PaymentEventType models.
  - 0016: Add currency field to Order model. Add a foreign key to the StockRecord model from the Line model.
  - 0017: Add a shipping_code field to the Order model.
  - 0018: ShippingAddress's phone_number is now a PhoneNumberField to allow better validation.
- Partner app:
  - 0008: Remove unnecessary partner_abstractstockalert table.

  - 0009: Create table for new PartnerAddress model.
  - 0010: Remove uniqueness constraint on product_id for the StockRecord model. This allows a product to have more than one stockrecord.
- Payment app:
  - 0002: Ensure all CharField fields are not nullable. This migration handles both the data- and schema-migration in one.
- Promotions app:
  - 0002: Ensure all CharField fields are not nullable.
  - 0003: Extend the max_length of the image field.
- Wishlist app:
  - 0001: Initial table creation

Features deprecated in 0.6

Accessing a product's stockrecords

Several properties and methods of the core *AbstractProduct* class have been deprecated following the change to allow multiple stockrecords per product.

- The has_stockrecord property no longer makes sense when there can be more than one stockrecord. It is replaced by *has_stockrecords*
- The stockrecord property is deprecated since it presumes there is only one stockrecord per product. Choosing the appropriate stockrecord is now the responsibility of the *strategy class*. A backward compatible version has been kept in place that selects the first stockrecord for a product. This will continue to work for sites that only have one stockrecord per product.

All availability logic has been moved to *availability policies* which are determined by the *strategy class*.

- The is_available_to_buy property is deprecated. The functionality is now part of availability policies.
- The is_purchase_permitted() method is deprecated. The functionality is now part of availability policies.

Checkout session manager

The shipping_method method of the *CheckoutSessionData* is now deprecated in favour of using shipping_method_code. It is being removed as the CheckoutSessionData class should only be responsible for retrieving data from the session, not loading shipping method instances.

Checkout order placement mixin

The following methods within *OrderPlacementMixin* are deprecated as the flow of placing an order has been changed.

- create_shipping_address_from_form_fields()
- create_shipping_address_from_user_address()
- create_user_address()

Bankcard model

The `card_number` is deprecated in favour of using `number`.

"Partner wrappers"

Before Oscar 0.6, availability and pricing logic was encapsulated in "partner wrappers". A partner wrapper was a class that provided availability and price information for a particular partner, as specified by the `OSCAR_PARTNER_WRAPPERS` setting. The stockrecord model had several properties and methods which delegated to the appropriate wrapper for the record's partner.

This functionality is now deprecated in favour of using *strategy classes*. How prices and taxes are determined is not generally a function of the partner, and so this system was not a good model. Strategy classes are much more flexible and allow better modelling of taxes and availability.

The following properties and methods from `StockRecord` are deprecated and will be removed for Oscar 0.7. These are all convenience properties and methods that delegate to the appropriate partner wrapper.

- `AbstractStockRecord.is_available_to_buy`
- `AbstractStockRecord.is_purchase_permitted`
- `AbstractStockRecord.availability_code`
- `AbstractStockRecord.availability`
- `AbstractStockRecord.max_purchase_quantity`
- `AbstractStockRecord.dispatch_date`
- `AbstractStockRecord.lead_time`
- `AbstractStockRecord.price_incl_tax`
- `AbstractStockRecord.price_tax`

All the above properties and methods have effectively been moved to the availability and pricing policies that a strategy class is responsible for loading. To upgrade your codebase, replace your partner wrapper classes with equivalent *availability and pricing policies*.

Test support extension brought back into core

The Oscar test support library has been ported back into Oscar core. This simplifies things and avoids circular dependency issues. If your project is using this extension, you should remove it as a dependency and use the analogous functionality from `oscar/test/`.

Oscar 0.6.1 release notes

This is Oscar 0.6.1. It fixes one potentially serious data loss issue and a few minor bugs.

Possible data loss from deleted users

Before this release, the foreign key from the `Order` model to the `User` model did not specify an `on_delete` behaviour. The default is for deletes to cascade to related objects, even if the field is nullable. Hence, deleting a user would also delete any orders they had placed.

As of 0.6.1, the foreign keys to user, shipping address and billing address on the `Order` model specify `on_delete=SET_NULL` to avoid orders being deleted accidentally.

See Django's docs for more info on `on_delete` options.

Missing translations

The 0.6 release failed to include several translations from Transifex due to a problem in the way we updated translation files before release. This release rectifies that and includes the latest translation files.

Known issues

- Django 1.4 only: The changes in #1127 mean you explicitly need to register a call to `migrate_alerts_to_users` when the `post_save` signal is emitted for a `User` model.

Bug fixes

The following bugs were fixed:

- #1109 - Workaround for a bug in Bootstrap regarding the collapsing of the navigation bar.
- #1121 - Added a confirmation view to removing products from wish lists because one can't POST to it in all cases.
- #1127 required that the `migrate_alerts_to_user` function is now explicitly called in Oscar's base User class. It previously was wired up as a `post_save` signal receiver on the User model, which does not work in Django 1.5+.
- #1128 - Calls to `Source.debit` without an `amount` argument were failing as `balance` was being called as a method instead of a property.
- #1130 - Variant products were not fetching the product class instance correctly within `is_shipping_required`.
- #1132 and #1149 - Rich text attributes were not supported. Should be displayed correctly now. Also introduced hooks for adding support for e.g. `file` and `image` types.
- #1133 - The order detail page for anonymous checkouts failed to render if reviews were disabled.
- #1134 - Fixed a bug caused where unicode characters in child products' titles were incorrectly handled.
- #1138 - Adjust the *`OrderAndItemCharges`* shipping method to not count lines that don't require shipping.
- #1146 - Various templates were adjusted to gracefully handle deleted products.

Oscar 0.6.2 release notes

This is Oscar 0.6.2. It fixes an unfortunate regression introduced in 0.6.1 as well as a couple of bugs.

Overriding models

Commit fa1f8403 changed the way signal receivers were registered. While this helped work around issues with the latest debug toolbar, it also broke the way custom models were imported. This happened as the relocated receiver imports caused core models to be imported before local ones.

This is fixed in 0.6.2 by reverting the original commit. Users of the debug toolbar are recommended to follow the explicit installation instructions to avoid any circular import issues that fa1f8403 was introduced to solve..

See #1159 for more details.

Bug fixes

The following bugs were fixed:

- #1157 - Ensure group products have a price submitted to the search backend when indexing.
- #1127 - Remove a circular dependency bug around importing the *StockAlert* model when indexing.

Oscar 0.6.3 release notes

This is Oscar 0.6.3. It fixes a few issues that have been discovered since the latest release.

Known issues

- Django 1.4 only: The changes in #1127 mean you explicitly need to register a call to `migrate_alerts_to_users` when the `post_save` signal is emitted for a `User` model.

Bug fixes

The following issues were fixed:

- Several strings have been marked translatable.
- #1167 - Offers without ranges can be created correctly.
- #1166, #1176 - Migrations work again with custom User model.
- #1186 - Fix bug with dashboard order search

Oscar 0.6.4 release notes

This is Oscar 0.6.4. This is a minor release which addresses a few niggles, mainly around how partner users are handled in the dashboard.

Bug fixes

The following issues were fixed:

- Editing variant products didn't correctly look up the parent product class.
- #1177 - Fix a regression in `get_classes` that prevented overridden dashboard apps being loaded correctly.
- #1273 - Dashboard partner views now allow user forms to be dynamically loaded (and hence overridden).
- #1275 - Dashboard partner user form now checks that the right fields are picked up from the user model (see also #1282, #1283)

Oscar 0.6.5 release notes

This is Oscar 0.6.5, a minor security release. If you rely on the `permissions_required` decorator or the `Application.permissions_map` and `Application.default_permissions` syntax, you must upgrade.

Bug fixes

- The `permissions_required` decorator now handles both methods and properties on the User model. Previously, it wasn't supported, but a docstring showed `is_anonymous` as an example, which is a method.

0.5 release branch

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.

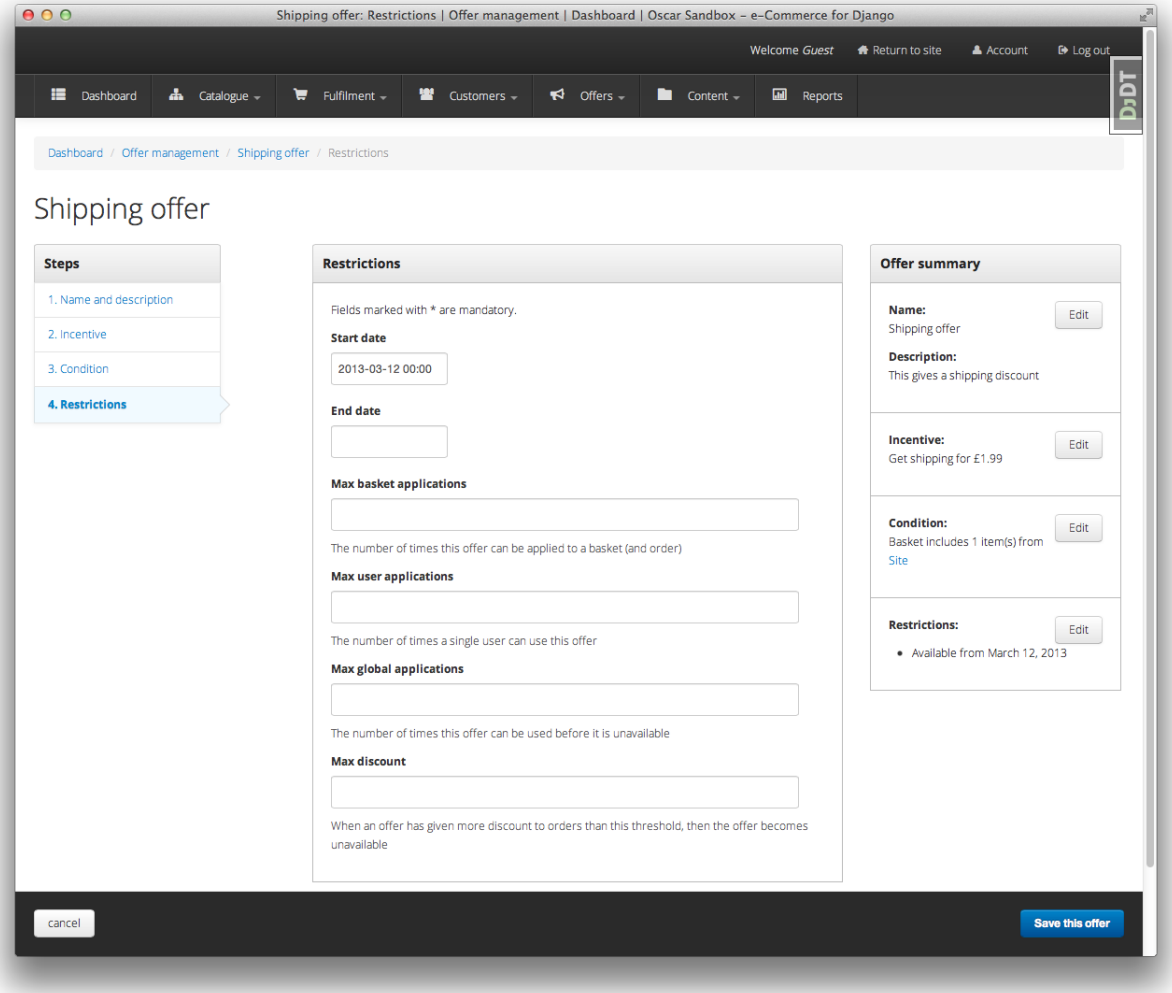

Fig. 1.6: The restrictions editing page for an offer within the dashboard.

- 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.

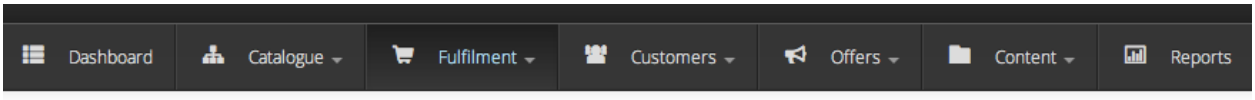

Fig. 1.7: The new dashboard navigation.

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).

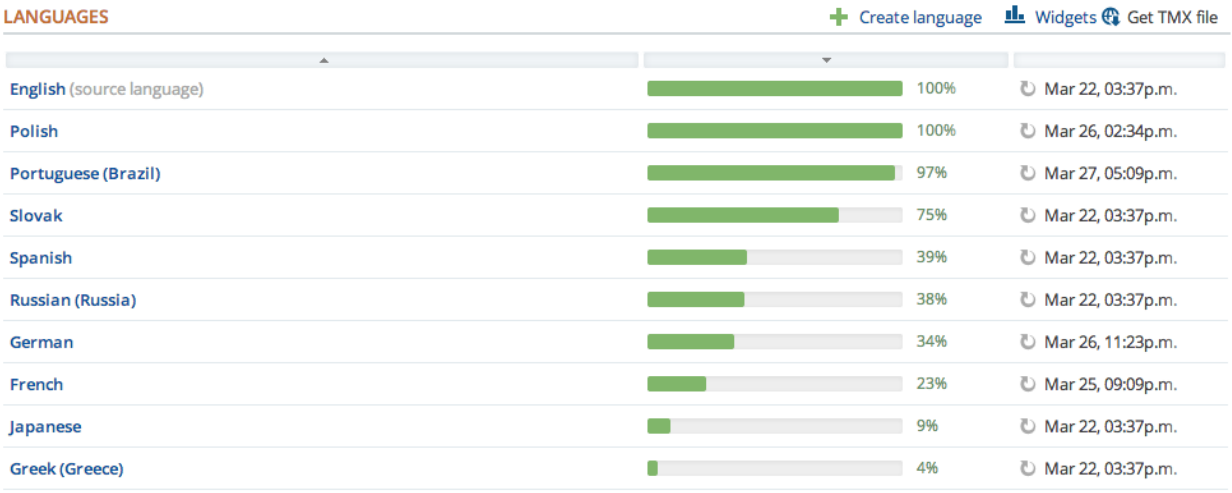

Fig. 1.8: A snippet from the Oscar Transifex page.

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).

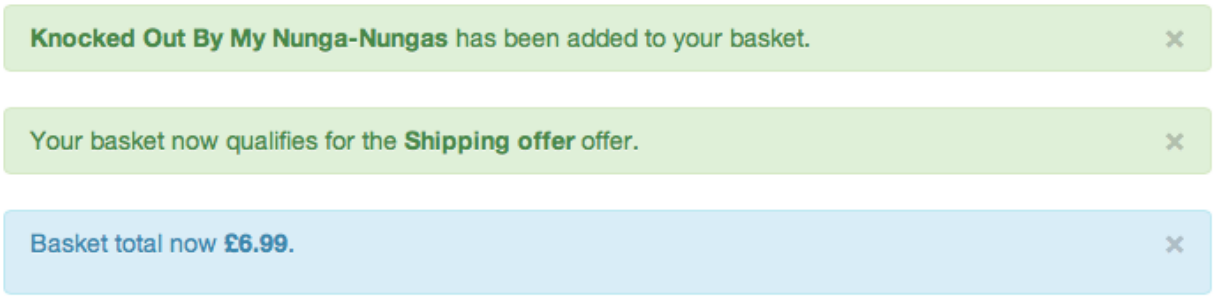

Fig. 1.9: A flash message indicating that the customer's basket has now qualified for a new offer.

- 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.

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

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.

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.

0.4 release branch

Oscar 0.4.11 release notes

release 2013-08-08

Add extra blocks to order dashboard template.

Oscar 0.4.10 release notes

release 2013-07-03

Extend range of bankcard expiry month field.

Oscar 0.4.9 release notes

release 2013-04-17

Make `AbstractStockRecord` abstract (*#645*)

Oscar 0.4.8 release notes

release 2013-04-08

Fix bug with order dashboard line editing (#622)

Oscar 0.4.7 release notes

release 2013-03-20

Fix bug with order dashboard searching (#587)

Oscar 0.4.6 release notes

release 2013-03-05

Fix dependencies in `setup.py`

Oscar 0.4.5 release notes

release 2013-01-25

Extend `get_class` to support loading from non-Oscar packages

Oscar 0.4.4 release notes

release 2013-01-16

Correct django-haystack in setup.py

Oscar 0.4.3 release notes

release 2013-01-16

Pin django-haystack version as backwards-incompatiable changes are happening

Oscar 0.4.2 release notes

release 2012-12-14

Nano-release to fix logout redirect bug

Oscar 0.4.1 release notes

release 2012-12-06

Nano-release to bump dependency versions.

Oscar 0.4.0 release notes

release 2012-10-19

Quite a big release this one. Several new features have been added since the 0.3 release series:

- Better support for digital products. Additional fields added to product class model.
- HTML editing within the dashboard
- A new email dashboard
- Major refactor of the offers module and test suite
- Product stock alerts: customers can request an alert when when a product comes back into stock
- Customer notifications: an API and inbox for sending users notifications

Upgrading

Four apps have new migrations. If you subclass these apps in your project, you will need to create a new schema migration for each to pick up the upstream changes.

- Basket:
  - A `price_excl_tax` has been added to `basket.Line`. This is useful for applications that use dynamic pricing where both the price with and without tax needs to be stored.
- Catalogue:
  - A `requires_shipping` field has been added to `catalogue.ProductClass` to facilitate better support for digital products (that don't require shipping).
  - The `code` field of `catalogue.Option` now has a unique index.
- Customer:
  - New models for stock alerts and notifications
  - The `email_subject_template` field from `customer.CommunicationEventType` is now nullable.
- Order:
  - An `offer_name` field has been added to `order.OrderDiscount` so retain audit information on discounts after offers are deleted.

Please ask on the mailing list if any other problems are encountered.

0.3 release branch

Oscar 0.3.7 release notes

release 2013-07-03

- Extend number of years in bankcard expiry field

Oscar 0.3.6 release notes

release 2013-04-08

- Fix line-handling bug in order dashboard.

Oscar 0.3.5 release notes

release 2012-09-28

A couple of minor adjustments for Tangent projects

- Add handling of custom redirect after adding to basket
- Add recursive URL decoration

Oscar 0.3.4 release notes

release 2012-09-24

- Rework price lookups in offer calculations (backport of functionality from 0.4)
- Add additional block to profile template

Diff: https://github.com/django-oscar/django-oscar/compare/0.3.3...0.3.4

Oscar 0.3.3 release notes

release 2012-08-24

- Minor bug fixes around category editing and order history.

Oscar 0.3.2 release notes

release 2012-08-13

- Bug fix for basket calculations.
- Bug fix for absolute discount benefit calculations.

Oscar 0.3.1 release notes

release 2012-08-08

- Now including the translation files.

Oscar 0.3.0 release notes

release 2012-08-08

- i18n support added - Oscar now ships with .po files for seven languages. Translation files are welcome.
- Category management added to dashboard
- Some improvements to how group/variant products are handled
- Improved installation process using makefile

Migrations

There are 3 new migrations in the catalogue app. If you have a local version, you will need to run:

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

to pick up the changes in Oscar's catalogue app.

Breaking changes

A new setting `OSCAR_MAIN_TEMPLATE_DIR` has been introduced as the template structure has been altered. This requires your `TEMPLATE_DIRS` setting to be altered to include this folder, eg:

```
from oscar import OSCAR_MAIN_TEMPLATE_DIR
TEMPLATE_DIRS = (
    location('templates'),
    OSCAR_MAIN_TEMPLATE_DIR
)
```

If you want to extend one of Oscar's templates, then use:

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

instead of:

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

0.2 release branch

Oscar 0.2.2 release notes

release 13 July 2012

Fixes a bug with applying absolute-discount benefits

Oscar 0.2.1 release notes

release 09 July 2012

Mainly small bug-fixes to templates and javascript.

Oscar 0.2.0 release notes

release 01 June 2012

Many components have been rewritten since 0.1 - Oscar is much more of a complete package now. New features include:

- Dashboard for managing catalogue, offers, stock, vouchers and more. This includes statistics pages to track performance.
- Sample templates, CSS and JS to get a shop up and running in a minutes.

- Updated documentation.
- Reworking of shipping methods.
- Automatic up-selling on the basket page. We now inform the user if they partially qualify for an offer.

The documentation still needs more work which we'll do over the next week or two.

0.1 release branch

Oscar 0.1.0 release notes

- Initial release - used in production by two major applications at Tangent but still quite rough around the edges. Many features were implemented directly in the applications rather than using a feature from oscar.
- Docs are a bit stale and need updating in 0.2

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!

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  # needs virtualenvwrapper
$ make install
```

If using Ubuntu, the `python-dev` package is required for some packages to compile.

The *sandbox* site can be used to examine changes locally. It is easily created by running:

```
$ make sandbox
```

JPEG Support

On Ubuntu, you need to install a few libraries to get JPEG support with Pillow:

```
$ sudo apt-get install python-dev libjpeg-dev libfreetype6-dev zlib1g-dev
```

If you already installed PIL (you did if you ran `make install` previously), reinstall it:

```
$ pip uninstall Pillow
$ pip install Pillow
```

Creating migrations

As the sandbox is a vanilla Oscar site, it is what we use to build migrations against:

```
$ make sandbox
$ sites/sandbox/manage.py schemamigration $YOURAPP --auto
```

Writing LESS/CSS

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

If you want to develop the LESS files, set:

```
OSCAR_USE_LESS = True
```

in `sites/sandbox/settings_local.py`. This will include the on-the-fly `less` pre-processor. That will allow you to see changes to the LESS files after a page reload.

You can manually compile the CSS files by running:

```
make css
```

For this to work, you will need to ensure that the pre-processor binary `lessc` is installed. Using npm, 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.

Testing migrations against MySQL and Postgres

To test the migrations against MySQL and Postgres you will need to set up an environment with both installed and do the following:

1. Change to sandbox folder and activate your virtualenv
2. Run helper script:

```
./test_migrations.sh

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

Reporting bugs and requesting features

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

- Be aware that Windows and the Django admin interface are unsupported; any tickets regarding that will get closed.
- 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 security issues

Security is paramount for e-commerce software like Oscar. Hence, we have adopted a policy which allows for responsible resporting and disclosure of security related issues.

If you believe you have found something in Oscar (or one of its extensions) which has security implications, please report is via email to `oscar.security@tangentlabs.co.uk`. Someone from the core team will acknowledge your report and take appropriate action.

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/`
- URL names use dashes not underscores.
- 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.

HTML

Please indent with four spaces.

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.
- Always submit pull requests from a custom branch. Don't submit from your master branch.

Test suite

Running tests

Oscar uses pytest to run the tests, which can be invoked using:

```
$ py.test
```

You can run a subset of the tests by passing a path:

    $ py.test tests/integration/offer/test_availability.py

To run an individual test class, use:

```
$ py.test tests/integration/offer/test_availability.py::TestASuspendedOffer
```

(Note the '::'.)

To run an individual test, use:

```
$ py.test tests/integration/offer/test_availability.py::TestASuspendedOffer::test_is_
↪unavailable
```

You can also run tests which match an expression via:

```
$ py.test tests/integration/offer/test_availability.py -k is_unavailable
```

Testing against different setups

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

```
$ detox
```

You need to have all Python interpreters to test against installed on your system. All other requirements are downloaded automatically. detox is a wrapper around tox, creating the environments and running the tests in parallel. This greatly speeds up the process.

Kinds of tests

Tests are split into 3 folders:

- 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

When running a subset of tests, Oscar uses the spec plugin. It is a good practice to name your test cases and methods so that the spec output reads well. For example:

```
$ py.test tests/integration/catalogue/test_product.py --spec
============================ test session starts ============================
platform darwin -- Python 3.6.0, pytest-3.0.6, py-1.4.33, pluggy-0.4.0
rootdir: /Users/sasha0/projects/djangooscar, inifile: setup.cfg
plugins: xdist-1.15.0, warnings-0.2.0, spec-1.1.0, django-3.1.2, cov-2.4.0
collected 15 items

tests/integration/catalogue/test_product.py::ProductCreationTests
    [PASS]  Allow two products without upc
    [PASS]  Create products with attributes
    [PASS]  None upc is represented as empty string
    [PASS]  Upc uniqueness enforced

tests/integration/catalogue/test_product.py::TopLevelProductTests
    [PASS]  Top level products are part of browsable set
    [PASS]  Top level products must have product class
    [PASS]  Top level products must have titles

tests/integration/catalogue/test_product.py::ChildProductTests
    [PASS]  Child products are not part of browsable set
    [PASS]  Child products dont need a product class
    [PASS]  Child products dont need titles
    [PASS]  Child products inherit fields

tests/integration/catalogue/test_product.py::TestAChildProduct
    [PASS]  Delegates requires shipping logic

tests/integration/catalogue/test_product.py::ProductAttributeCreationTests
    [PASS]  Entity attributes
    [PASS]  Validating option attribute

tests/integration/catalogue/test_product.py::ProductRecommendationTests
```

```
    [PASS]  Recommended products ordering

========================== 15 passed in 15.39 seconds ========================
```

Writing documentation

Directory Structure

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/`.

Style guides

Oscar currently does not have it's own style guide for writing documentation. Please carefully review style guides for Python and Django. Please use gender-neutral language.

Python Module Index

o

Index

A

B

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W