djangoSHOP

Release 0.9.1

November 17, 2016

This is the documentation starting from version 0.9; if you are looking for the documentation of django-shop version 0.2, please check the sidebar of RTD.

Version 0.9 of **djangoSHOP** is a complete rewrite of the code base, keeping the concepts of model overriding and cart modifiers. With some effort it should be possible to migrate existing projects to this new release.

CHAPTER 1

Software Architecture

The **djangoSHOP** framework is, as its name implies, a framework and not a software which runs out of the box. Instead, an e-commerce site built upon **djangoSHOP**, always consists of this framework, a bunch of other Django apps and the **merchant's own implementation**. While this may seem more complicate than a ready-to-use solution, it gives the programmer enormous advantages during the implementation:

Not everything can be "explained" to a software system using user interfaces. When reaching a certain point of complexity, it normally is easier to pour those requirements into code, rather than to expect yet another set of configuration buttons.

When evaluating **djangoSHOP** with other e-commerce solutions, I therefore suggest to do the following litmus test:

Consider a product which shall be sold world-wide. Depending on the country's origin of the request, use the native language and the local currency. Due to export restrictions, some products can not be sold everywhere. Moreover, in some countries the value added tax is part of the product's price, and must be stated separately on the invoice, while in other countries, products are advertised using net prices, and tax is added later on the invoice.

Instead of looking for software which can handle such a complex requirement, rethink about writing your own plugins, able to handle this. With the **django**, **REST** and **djangoSHOP** frameworks, this normally is possible in a few dozen lines of clearly legible Python code. Compare this to solutions, which claim to handle such complex requirements. They normally are shipped containing huge amounts of features, which very few merchants ever require, but which bloat the overall system complexity, making such a piece of software expensive to maintain.

1.1 Design Decisions

1.1.1 Single Source of Truth

A fundamental aspect of good software design is to follow the principle of "Don't repeat yourself", often denoted as DRY. In **djangoSHOP** we aim for a *single source of truth*, wherever possible.

For instance have a look at the `shop.models.address.BaseShippingAddress`. Whenever we add, change or remove a field, the ORM mapper of Django gets notified and with `./manage.py makemigrations` followed by `./manage.py migrate` our database scheme is updated. But even the input fields of our address form adopt to all changes in our address model. Even the client side form field validation adopts to every change in our address model. As we can see, here our single source of truth is the address model.

1.1.2 Feature Completeness

A merchant who wants to implement a unique feature for his e-commerce site, *must* never have to touch the code of the framework. Aiming for ubiquity means, that no matter how challenging a feature is, it *must be possible to be* *implemented* into the merchant's own implementation, rather than by patching the framework itself.

Otherwise *this framework contains a bug* - not just a missing feature! I'm sure some merchants will come up with really weird ideas, I never have thought of. If the **djangoSHOP** framework inhibits to add a feature, then feel free to create a bug report. The claim "*feature completeness*" for a framework is the analogue to the term "*Turing completeness*" for programming languages.

Consider that on many sites, a merchant's requirement is patched into existing code. This means that every time a new version of the e-commerce software is released, that patch must be repeatedly adopted. This can become rather dangerous when security flaws in that software must be closed immediately. **DjangoSHOP** instead is designed, so that the merchant's implementation and third party plugins have to subclass its models and to override its templates accordingly.

1.1.3 Minimalism

In a nutshell, **djangoSHOP** offers this set of basic functionalities, to keep the framework simple and stupid (KISS) without reinventing the wheel:

- A catalog to display product lists and detail views.
- Some methods to add products to the cart.
- A way to remove items from the cart or change their quantities.
- A set of classes to modify the cart's totals.
- A collection of forms, where customers can add personal, shipping and payment information.
- A way to perform the purchase: this converts the cart into an order.
- A list view where customers can lookup their previously performed orders
- A backend tool which helps to track the state of orders.

All functionality required to build a real e-commerce site, sits on top of this. Computing taxes for instance, can vary a lot among different legislations and thus is not part of the framework. The same applies for vouchers, rebates, delivery costs, etc.

These are the parts, which must be fine tuned by the merchant. They can be rather complicate to implement and are best implemented by separate plugins.

1.1.4 Separation of Concern

Compared to other e-commerce solutions, the **djangoSHOP** framework has a rather small footprint in terms of code lines, database tables and classes. This does not mean, that its functionality is somehow limited. Instead, the merchant's own implementation can become rather large. This is because **djangoSHOP** implies dependencies to many third party Django apps.

Having layered systems gives us programmers many advantages:

- We don't have to reinvent the wheel for every required feature.
- Since those dependencies are used in other applications, they normally are tested quite well.
- No danger to create circular dependencies, as found often in big libraries and stand alone applications.
- Better overview for newcomers, which part of the system is responsible for what.
- Easier to replace one component against another one.

Fortunately Django gives us all the tools to stitch those dependencies together. If for instance we would use one of the many PHP-based e-commerce system, we'd have to stay inside their modest collection for third party apps, or reinvent the wheel. This often is a limiting factor compared to the huge ecosystems arround Django.

1.1.5 Inversion of Control

Wherever possible, **djangoSHOP** tries to delegate the responsibility for taking decision to the merchant's implementation of the site. Let explain this by a small example: When the customer adds a product to the cart, **djangoSHOP** consults the implementation of the product to determine whether the given item is already part of the cart or not. This allows the merchant's implementation to fine tune its product variants.

1.2 Core System

Generally, the shop system can be seen in three different phases:

1.2.1 The shopping phase

From a customers perspective, this is where we look around at different products, presumably in different categories. We denote this as the catalog list- and catalog detail views. Here we browse, search and filter for products. In one of the list views, we edit the quantity of the products to be added to our shopping cart.

Each time a product is added, the cart is updated which in turn run the so named "Cart Modifiers". Cart modifiers sum up the line totals, add taxes, rebates and shipping costs to compute the final total. The Cart Modifiers are also during the checkout phase (see below), since the chosen shipping method and destination, as well as the payment method may modify the final total.

1.2.2 The checkout process

Her the customer must be able to refine the cart' content: Change the quantity of an item, or remove that item completely from the cart.

During the checkout process, the customer must enter his addresses and payment informations. These settings may also influence the cart's total.

The final step during checkout is the purchase operation. This is where the cart's content is converted into an order object and emptied afterwards.

1.2.3 The fulfillment phase

It is now the merchants's turn to take further steps. Depending on the order status, certain actions must be performed immediately or the order must be kept in the current state until some external events happen. This could be a payment receivement, or that an ordered item arrived in stock. While setting up a **djangoSHOP** project, the allowed status transitions for the fulfillment phase can be plugged together, giving the merchant the possibility to programmatically define his order workflows.

1.3 Plugins

Django SHOP defines 5 types of different plugins:

1. Product models
2. Cart modifiers
3. Payment backends
4. Shipping backends
5. Order workflow modules

They may be added as a third party **djangoSHOP** plugin, or integrated into the merchant's implementation.

CHAPTER 2

Unique Features of djangoSHOP

2.1 djangoSHOP requires to describe your products instead of prescribing prefabricated models

Products can vary wildly, and modeling them is not always trivial. Some products are salable in pieces, while others are continues. Trying to define a set of product models, capable for describing all such scenarios is impossible – describe your product by customizing the model and not vice versa.

2.1.1 E-commerce solutions, claiming to be plug-and-play, normally use one of these (anti-)patterns

Either, they offer a field for every possible variation, or they use the Entity-Attribute-Value pattern to add meta-data for each of your models. This at a first glance seems to be easy. But both approaches are unwieldy and have serious drawbacks. They both apply a different "physical schema" – the way data is stored, rather than a "logical schema" – the way users and applications require that data. As soon as you have to combine your e-commerce solution with some Enterprise-Resource-Planning software, additional back-and-forward conversion routines have to be added.

2.1.2 In djangoSHOP, the physical representation of a product corresponds to its logical

djangoSHOP's approach to this problem is to have minimal set of models. These abstract models are stubs provided to subclass the physical models. Hence the logical representation of the product conforms to their physical one. Moreover, it is even possible to represent various types of products by subclassing polymorphically from an abstract base model. Thanks to the Django framework, modeling the logical representation for a set of products, together with an administration backend, becomes almost effortless.

2.2 djangoSHOP is multilingual

Products offered in various regions, normally require attributes in different natural languages. For such a set of products, these attributes can be easily modelled using translatable fields. This lets you seamlessly built a multilingual e-commerce site.

2.3 djangoSHOP supports multiple currencies

djangoSHOP is shipped with a set of currency types, bringing their own money arithmetic. This adds an additional layer of security, because one can not accidentally sum up different currencies. These money types always know how to represent themselves in different local environments, prefixing their amount with the correct currency symbol. They also offer the special amount "no price" (represented by −), which behaves like zero but is handy for gratuitous items.

2.4 djangoSHOP directly plugs into djangoCMS

Product detail pages may use all templatetags from djangoCMS, such as the `{% placeholder ... %}`, the `{% static_placeholder ... %}`, or other CMS tags.

djangoSHOP does not presuppose categories to organize product list views. Instead djangoCMS pages can be specialized to handle product lists via a CMS app. This allows the merchant to organize products into categories, using the existing page hierarchy from the CMS. It also allows to offer single products from a CMS page, without requiring any category.

2.5 djangoSHOP is based on REST

- djangoSHOP uses the Django REST framework and hence does not require any Django View
- Every view is based on REST interfaces.
- Infinite scrolling and paginated listings use the same template.
- Views for cart, checkout etc. can be inserted into exiting pages.
- This means that one can navigate through products, add them to the cart, modify the cart, register himself as new customer (or proceed as guest), add his shipping information, pay via Stripe and view his past orders. Other Payment Service Providers can be added in a pluggable manner.

Every page in the shop: product-list, product-detail, cart, checkout-page, orders-list, order-detail etc. is part of the CMS and can be edited through the plugin editor. The communication between the client and these pages is done exclusively through REST. This has the nice side-effect, that the merchants shop implementation does not require any Django-View.

djangoSHOP is shipped with individual components for each task. These plugins then can be placed into any CMS placeholder using the plugin editor. Each of these plugins is shipped with their own overridable template, which can also be used as a stand-alone template outside of a CMS placeholder. Templates for bigger tasks, such as the Cart-View are granular, so that the HTML can be overridden partially.

Authentication is done through auth-rest, which allows to authenticate against a bunch of social networks, such as Google+, Facebook, GitHub, etc in a pluggable manner.

Moreover, the checkout process is based on a configurable finite state machine, which means that a merchant can adopt the shops workflow to the way he is used to work offline.

Client code is built using Bootstrap-3.3 and AngularJS-1.3. jQuery is required only for the backends administration interface. All browser components have been implemented as AngularJS directives, so that they can be reused between projects. For instance, my current merchant implementation does not have a single line of customized JavaScript.

This makes is very easy, even for non-programmers, to implement a shop. A merchant only has to adopt his product models, optionally the cart and order models, and override the templates.

CHAPTER 3

Tutorial

This tutorial shows how to setup a working e-commerce site with **djangoSHOP** using the given dependencies. The code required to setup this demo can be found in the example/myshop folder.

3.1 Tutorial

3.1.1 Introduction

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

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

3.1.2 Prepare the Installation

To run the examples shown in this tutorial, you must install **django-shop** from GitHub, since the pip-installable from PyPI only contains the main files. Before proceeding, please make sure virtualenv is installed on your system, otherwise you would pollute your Python site-packages folder.

Also ensure that these packages are installed using the favorite package manager of your operating system:

- Python 2.7
- Redis: http://redis.io/
- SQLite: https://www.sqlite.org/
- bower: http://bower.io/
- Node Package Manager: https://www.npmjs.com/
- Python 2.7 (Latest minor version recommended)
- Django 1.9 (Latest minor version recommended)

```
$ virtualenv shoptutorial
$ source shoptutorial/bin/activate
$ mkdir Tutorial; cd Tutorial
(shoptutorial)$ git clone --depth 1 https://github.com/awesto/django-shop
(shoptutorial)$ cd django-shop
(shoptutorial)$ pip install -e .
(shoptutorial)$ pip install -r requirements/tutorial.txt
```

```
(shoptutorial)$ npm install
(shoptutorial)$ bower install
```

these statements will setup an environment, which runs a demo shop out of the box.

You may populate the database with your own products, or if impatient, *Quickstart a Running Demo* using prepared CMS page layouts, products and media files.

Create a database for the demo

Finally we must create a database to run our example project:

```
(shoptutorial)$ cd example
(shoptutorial)$ export DJANGO_DEBUG=1
(shoptutorial)$ ./manage.py migrate
(shoptutorial)$ ./manage.py createsuperuser
Email address: admin@example.org
Username: admin
Password:
Password (again):
Superuser created successfully.
(shoptutorial)$ ./manage.py runserver
```

Finally point a browser onto http://localhost:8000/ and log in as the superuser you just created.

3.1.3 Add some pages to the CMS

In **djangoSHOP**, every page, with the exception of the product's detail pages, can be rendered by the CMS. Therefore, unless you need a special landing page, start immediately with the *Catalog List View* of your products. Change into the Django Admin backend, chose the section

Start > django CMS > Pages

and add a Page. As its **Title** chose "Smart Cards". Then change into the **Advanced Settings** at the bottom of the page. In this editor window, locate the field **Application** and select **Products List**. Then save the page and click on **View on site**.

Now change into **Structure** mode and locate the placeholder named **Main content container**. Add a plugin from section **Bootstrap** named **Row**. Below that Row add a Column with a width of 12 units. Finally, below the last Column add a plugin from section **Shop** named **Catalog List View**.

Now we have a working catalog list view, but since we havn't added any products to the database yet, we won't see any items on our page.

3.2 Quickstart a Running Demo

3.2.1 Using a Docker image

To get a first impression of the **djangoSHOP** examples, you may use a prepared Docker container. If not already available on your workstation, first install the Docker runtime environment and start a Docker machine.

Now you may run a fully configured **djangoSHOP** image on your local machine:

```
docker run -p 9001:9001 jrief/uwsgi-django-shop:latest
```

This image is rather large (1.7 GB) therefore it may take some time to download.

Locate the IP address of the running container using `docker-machine ip default`. Then point a browser onto this address using port 9001, for instance http://192.168.99.100:9001/en/

Please note that before bootstrapping, a full text index is built and the images are thumbnailed. This takes additional time. Therefore, if you stop the running container, before rerunning the Docker image it is recommended to restart the container. First locate it using

```
$ docker ps -a
CONTAINER ID  IMAGE                             COMMAND                 CREATED
79b7b69a7473  jrief/uwsgi-django-shop:latest   "/usr/sbin/uwsgi --in"  11 minutes ago
...
$ docker start 79b7b69a7473
```

and then restart it. The access the administration backed, sign in as user "*admin*" with password "*secret*".

Note: This demo does not function with the Payment Service Provider Stripe, because each merchant requires its own credentials. The same applies for sending emails, because the application requires an outgoing SMTP server.

3.2.2 The classic approach

Alternatively you may also download all dependencies and start the project manually. If you want to use the demo as a starting point, this probably is the better solution.

Filling your CMS with page content and adding products is a boring job. Impatient users may start three demos using some prepared sample data. First assure that all dependencies are installed into its virtual environment as described in section "*Prepare the Installation*". Then instead of adding pages and products manually, download the media files and unpack them into the folder `django-shop`:

```
(shoptutorial)$ tar zxf DOWNLOAD/FOLDER/django-shop-workdir.tar.gz
```

Starting from this folder, you can run all three demos: The first, simple demo shows how to setup a monolingual shop, with one product type. The second, internationalized demo shows how to setup a multilingual shop, with one product type. For translation of model attributes, this installation uses the django-parler app. The third, polymorphic demo shows how to setup a shop with many different product types. To handle the polymorphism of products, this installation uses the django-polymorphic app.

Note: All demos can be started independently from each other, but you are encouraged to start with the "Simple Product", and then proceed to the more complicate examples.

3.2.3 Simple Product Demo

Assure you are in the `django-shop` folder and using the correct virtual environment. Then in a shell invoke:

```
(shoptutorial)$ cd example
(shoptutorial)$ export DJANGO_SHOP_TUTORIAL=simple DJANGO_DEBUG=1
(shoptutorial)$ ./manage.py migrate
(shoptutorial)$ ./manage.py loaddata fixtures/myshop-simple.json
(shoptutorial)$ ./manage.py runserver
```

Point a browser onto http://localhost:8000/admin/ and sign in as user "*admin*" with password "*secret*".

This runs the demo for *Modeling a simple product*.

3.2.4 Internationalized Products

In this demo the description of the products can be translated into different natural languages.

When migrating from the Simple Products demo, assure you are in the `django-shop` folder and using the correct virtual environment. Then in a shell invoke:

```
(shoptutorial)$ cp workdir/db-simple.sqlite3 workdir/db-i18n.sqlite3
(shoptutorial)$ cd example
(shoptutorial)$ export DJANGO_SHOP_TUTORIAL=i18n DJANGO_DEBUG=1
(shoptutorial)$ ./manage.py migrate
(shoptutorial)$ ./manage.py runserver
```

Alternatively, if you prefer to start with an empty database, assure that the file `workdir/db-i18n.sqlite3` is missing. Then in a shell invoke:

```
(shoptutorial)$ cd example
(shoptutorial)$ export DJANGO_SHOP_TUTORIAL=i18n DJANGO_DEBUG=1
(shoptutorial)$ ./manage.py migrate
(shoptutorial)$ ./manage.py loaddata fixtures/myshop-i18n.json
(shoptutorial)$ ./manage.py runserver
```

Point a browser onto http://localhost:8000/admin/ and sign in as user "*admin*" with password "*secret*".

This runs a demo for *Modeling a Multilingual Product*.

3.2.5 Polymorphic Products

In this demo we show how to handle products with different properties and in different natural languages. This example can't be migrated from the previous demos, without loosing lots of information. It is likely that you don't want to add the Smart Phones manually, it is suggested to start using a fixture.

This example shows how to add Smart Phones in addition to the existing Smart Cards. Assure you are in the `django-shop` folder and using the correct virtual environment. Then in a shell invoke:

```
(shoptutorial)$ rm workdir/db-polymorphic.sqlite3
(shoptutorial)$ cd example
(shoptutorial)$ export DJANGO_SHOP_TUTORIAL=polymorphic
(shoptutorial)$ ./manage.py migrate
(shoptutorial)$ ./manage.py loaddata fixtures/myshop-polymorphic.json
(shoptutorial)$ ./manage.py runserver
```

Point a browser onto http://localhost:8000/admin/ and sign in as user "*admin*" with password "*secret*".

This runs a demo for *Products with Different Properties*.

3.3 Modeling a simple product

As a simple example, this tutorial uses Smart Cards as its first product. As emphasized in section tutorial/customer-model, **djangoSHOP** is not shipped with ready to use product models. Instead the merchant must declare these models based on the products properties. Lets have a look ar a model describing a typical Smart Card:

Listing 3.1: myshop/models/simple/smartcard.py

```python
from djangocms_text_ckeditor.fields import HTMLField
from shop.money.fields import MoneyField
from shop.models.product import BaseProduct, BaseProductManager
from shop.models.defaults.mapping import ProductPage, ProductImage
@python_2_unicode_compatible
class SmartCard(BaseProduct):
    # common product fields
    product_name = models.CharField(max_length=255, verbose_name=_("Product Name"))
    slug = models.SlugField(verbose_name=_("Slug"))
    unit_price = MoneyField(_("Unit price"), decimal_places=3,
        help_text=_("Net price for this product"))
    description = HTMLField(verbose_name=_("Description"),
    images = models.ManyToManyField('filer.Image', through=ProductImage)

```

Here our model `SmartCard` inherits directly from `BaseProduct`, which is a stub class, hence the most common fields, such as `product_name`, `slug` and `unit_price` must be added to our product here. Later on we will see why these fields, even though required by each product, can not be part of our abstract model `BaseProduct`.

Additionally a smart card has some product specific properties:

```python
                                help_text=_("Description for the list view of products."))

    # product properties
    manufacturer = models.ForeignKey(Manufacturer, verbose_name=_("Manufacturer"))
    CARD_TYPE = (2 * ('{}{}'.format(s, t),)
                     for t in ('SD', 'SDXC', 'SDHC', 'SDHC II') for s in ('', 'micro '))
    card_type = models.CharField(_("Card Type"), choices=CARD_TYPE, max_length=15)
    SPEED = ((str(s), "{} MB/s".format(s)) for s in (4, 20, 30, 40, 48, 80, 95, 280))
    speed = models.CharField(_("Transfer Speed"), choices=SPEED, max_length=8)
    product_code = models.CharField(_("Product code"), max_length=255, unique=True)
    storage = models.PositiveIntegerField(_("Storage Capacity"),
        help_text=_("Storage capacity in GB"))
```

these class attributes depend heavily on the data sheet of the product to sell.

Finally we also want to position our products into categories and sort them:

```python
        help_text=_("Storage capacity in GB"))

    # controlling the catalog
    order = models.PositiveIntegerField(verbose_name=_("Sort by"), db_index=True)
    cms_pages = models.ManyToManyField('cms.Page', through=ProductPage,
        help_text=_("Choose list view this product shall appear on."))
    images = models.ManyToManyField('filer.Image', through=ProductImage)
```

The field `order` is used to keep track on the sequence of our products while rendering a list view.

The field `cms_pages` specifies on which pages of the CMS a product shall appear.

Note: If categories do not require to keep any technical properties, it often is completely sufficient to use CMS pages as their surrogates.

Finally `images` is another many-to-many relation, allowing to associate none, one or more images to a product.

Both fields `cms_pages` and `images` must use the through parameter. This is because we have two many-to-many mapping tables which are part of the merchant's project rather than the **djangoSHOP** application. The first of those mapping tables has foreign keys onto the models `cms.Page` and `myshop.SmartCard`. The second table has foreign keys onto the models `filer.Image` and `myshop.SmartCard` again. Since the model `myshop.SmartCard` has been declared by the merchant himself, he also is responsible for managing those many-to-many mapping tables.

Additionally each product model requires these attributes:

- A model field or property method named `product_name`: It must returns the product's name in its natural language.
- A method `get_price(request)`: Returns the product price. This can depend on the given region, which is available through the request object.
- A method `get_absolute_url()`: Returns the canonical URL of a product.
- The `object` attribute must be of type `BaseProductManager` or derived from thereof.

These product model attributes are optional, but highly recommended:

- A model field or property method named `product_code`: It shall returns a language independent product code or article number.
- A property method `sample_image`: It shall returns a sample image for the given product.

3.3.1 Add Model `myshop.SmartCard` to Django Admin

For reasons just explained, it is the responsibility of the project to manage the many-to-many relations between its CMS pages and the images on one side, and the product on the other side. Therefore we can't use the built-in admin widget `FilteredSelectMultiple` for these relations.

Instead **djangoSHOP** is shipped with a special mixin class `CMSPageAsCategoryMixin`, which handles the relation between CMS pages and the product. This however implies that the field used to specify this relation is named `cms_pages`.

```python
from adminsortable2.admin import SortableAdminMixin
from shop.admin.product import CMSPageAsCategoryMixin, ProductImageInline
from myshop.models import SmartCard


@admin.register(SmartCard)
class SmartCardAdmin(SortableAdminMixin, CMSPageAsCategoryMixin, admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': ('product_name', 'slug', 'product_code', 'unit_price', 'active', 'description',
        }),
        (_("Properties"), {
            'fields': ('manufacturer', 'storage', 'card_type', 'speed',)
        }),
    )
    inlines = (ProductImageInline,)
    prepopulated_fields = {'slug': ('product_name',)}
    list_display = ('product_name', 'product_code', 'unit_price', 'active',)
    search_fields = ('product_name',)
```

For images, the admin class must use a special inline class named `ProductImageInline`. This is because the merchant might want to arrange the order of the images and therefore a simple `SelectMultiple` widget won't do this job here.

Extend our simple product to support other natural languages by *Modeling a Multilingual Product*.

3.4 Modeling a Multilingual Product

Let's extend our previous `SmartCard` model to internationalize our shop site. Normally the name of a Smart Card model is international anyway, say "*Ultra Plus micro SDXC*", so it probably won't make much sense to use a translatable field here. The model attribute which certainly makes sense to be translated into different languages, is the `description` field.

3.4.1 Run the Multilingual Demo

To test this example, set the shell environment variable `export DJANGO_SHOP_TUTORIAL=i18n`, then apply the modified models to the database schema:

```
./manage.py migrate myshop
```

Alternatively recreate the database as explained in *Create a database for the demo*.

Afterwards start the demo server:

```
./manage.py runserver
```

3.4.2 The Multilingal Product Model

DjangoSHOP uses the library django-parler for model translations. We therefore shall rewrite our model as:

Listing 3.2: myshop/models/i18n/smartcard.py

```python
from djangocms_text_ckeditor.fields import HTMLField
from parler.managers import TranslatableManager, TranslatableQuerySet
from parler.models import TranslatableModel, TranslatedFieldsModel
from parler.fields import TranslatedField
from polymorphic.query import PolymorphicQuerySet
from shop.money.fields import MoneyField
from shop.models.product import BaseProductManager, BaseProduct
from shop.models.defaults.mapping import ProductPage, ProductImage
from myshop.models.properties import Manufacturer

class ProductQuerySet(TranslatableQuerySet, PolymorphicQuerySet):
    pass

class ProductManager(BaseProductManager, TranslatableManager):
    queryset_class = ProductQuerySet
@python_2_unicode_compatible
class SmartCard(BaseProduct, TranslatableModel):
    product_name = models.CharField(max_length=255, verbose_name=_("Product Name"))
    slug = models.SlugField(verbose_name=_("Slug"))
    unit_price = MoneyField(_("Unit price"), decimal_places=3,
        help_text=_("Net price for this product"))
    images = models.ManyToManyField('filer.Image', through=ProductImage)



class SmartCardTranslation(TranslatedFieldsModel):
    master = models.ForeignKey(SmartCard, related_name='translations',
        null=True)
    description = HTMLField(verbose_name=_("Description"),
        help_text=_("Description for the list view of products."))

    class Meta:
        unique_together = [('language_code', 'master')]
```

In comparison to the simple Smart Card model, the field `description` can now accept text in different languages.

In order to work properly, a model with translations requires an additional model manager and a table storing the translated fields. Accessing an instance of this model behaves exactly the same as an untranslated model. Therefore it can be used as a drop-in replacement for our simple `SmartCard` model.

3.4.3 Translatable model in Django Admin

The admin requires only a small change. Its class must additionally inherit from `TranslatableAdmin`. This adds a tab for each configured language to the top of the detail editor. Therefore it is recommended to group all multilingual fields into one fieldset to emphasize that these fields are translatable.

Listing 3.3: myshop/admin/i18n/smartcard.py

```
from django.contrib import admin
from django.utils.translation import ugettext_lazy as _
from adminsortable2.admin import SortableAdminMixin
from parler.admin import TranslatableAdmin
from shop.admin.product import CMSPageAsCategoryMixin, ProductImageInline
from myshop.models import SmartCard

@admin.register(SmartCard)
class SmartCardAdmin(SortableAdminMixin, TranslatableAdmin,
                     CMSPageAsCategoryMixin, admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': ('product_name', 'slug', 'product_code', 'unit_price', 'active',),
        }),
        (_("Translatable Fields"), {
            'fields': ('description',)
        }),
        (_("Properties"), {
            'fields': ('manufacturer', 'storage', 'card_type',)
        }),
    )
    inlines = (ProductImageInline,)
    prepopulated_fields = {'slug': ('product_name',)}
    list_display = ('product_name', 'product_code', 'unit_price', 'active',)
    search_fields = ('product_name',)
```

Extend our discrete product type, to polymorphic models which are able to support many different product types: *Products with Different Properties*.

3.5 Products with Different Properties

In the previous examples we have seen that we can model our products according to their physical properties, but what if we want to sell another type of a product with different properties. This is where polymorphism enters the scene.

3.5.1 Run the Polymorphic Demo

To test this example, set the shell environment variable `export DJANGO_SHOP_TUTORIAL=polymorphic`, then recreate the database as explained in *Create a database for the demo* and start the demo server:

```
./manage.py runserver
```

3.5.2 The Polymorphic Product Model

If in addition to Smart Cards we also want to sell Smart Phones, we must declare a new model. Here instead of duplicating all the common fields, we unify them into a common base class named `Product`. Then that base class shall be extended to become either our known model `SmartCard` or a new model `SmartPhone`.

To enable polymorphic models in **djangoSHOP**, we require the application django-polymorphic. Here our models for Smart Cards or Smart Phones will be split up into a generic part and a specialized part. The generic part goes into our new `Product` model, whereas the specialized parts remain in their models.

You should already start to think about the layout of the list views. Only attributes in model `Product` will be available for list views displaying Smart Phones side by side with Smart Cards. First we must create a special Model Manager which unifies the query methods for translatable and polymorphic models:

Listing 3.4: myshop/models/i18n/polymorphic/product.py

```python
from djangocms_text_ckeditor.fields import HTMLField
from parler.models import TranslatableModel, TranslatedFieldsModel
from parler.fields import TranslatedField
from parler.managers import TranslatableManager, TranslatableQuerySet
from polymorphic.query import PolymorphicQuerySet
from shop.models.product import BaseProductManager, BaseProduct
from shop.models.defaults.mapping import ProductPage, ProductImage
from myshop.models.properties import Manufacturer

class ProductQuerySet(TranslatableQuerySet, PolymorphicQuerySet):
    pass

class ProductManager(BaseProductManager, TranslatableManager):
    queryset_class = ProductQuerySet
@python_2_unicode_compatible
class Product(BaseProduct, TranslatableModel):
    product_name = models.CharField(max_length=255, verbose_name=_("Product Name"))
    slug = models.SlugField(verbose_name=_("Slug"), unique=True)
    description = TranslatedField()

```

The next step is to identify which model attributes qualify for being part of our Product model. Unfortunately, there is no silver bullet for this problem and that's one of the reason why **djangoSHOP** is shipped without any prepared model for it. If we want to sell both Smart Cards and Smart Phones, then this Product model may do its jobs:

Listing 3.5: myshop/models/i18n/polymorphic/product.py

```python

    # common product properties
    manufacturer = models.ForeignKey(Manufacturer, verbose_name=_("Manufacturer"))

    # controlling the catalog
    order = models.PositiveIntegerField(verbose_name=_("Sort by"), db_index=True)
    cms_pages = models.ManyToManyField('cms.Page', through=ProductPage,
        help_text=_("Choose list view this product shall appear on."))
    images = models.ManyToManyField('filer.Image', through=ProductImage)
```

Model for Smart Card

The model used to store translated fields is the same as in our last example. The new model for Smart Cards now inherits from Product:

Listing 3.6: myshop/models/i18n/polymorphic/smartcard.py

```python
from django.db import models
from django.utils.translation import ugettext_lazy as _
from shop.money.fields import MoneyField
from .product import Product

class SmartCard(Product):
    # common product fields
    unit_price = MoneyField(_("Unit price"), decimal_places=3,
        help_text=_("Net price for this product"))

    # product properties
    CARD_TYPE = (2 * ('{}{}'.format(s, t),)
                 for t in ('SD', 'SDXC', 'SDHC', 'SDHC II') for s in ('', 'micro '))
    card_type = models.CharField(_("Card Type"), choices=CARD_TYPE, max_length=15)
    SPEED = ((str(s), "{} MB/s".format(s)) for s in (4, 20, 30, 40, 48, 80, 95, 280))
    speed = models.CharField(_("Transfer Speed"), choices=SPEED, max_length=8)
    product_code = models.CharField(_("Product code"), max_length=255, unique=True)
    storage = models.PositiveIntegerField(_("Storage Capacity"),
        help_text=_("Storage capacity in GB"))

```

Model for Smart Phone

The product model for Smart Phones is intentionally a little bit more complicated. Not only does it have a few more attributes, but Smart Phones can be sold with different specifications of internal storage. The latter influences the price and the product code. This is also the reason why we didn't move the model fields `unit_price` and `products_code` into our base class `Product`, although every product in our shop requires them.

When presenting Smart Phones in our list views, we want to focus on different models, but not on each flavor, ie. its internal storage. Therefore customers will have to differentiate between the concrete Smart Phone variations, whenever they add them to their cart, but not when viewing them in the catalog list. For a customer, it would be very boring to scroll through lists with many similar products, which only differentiate by a few variations.

This means that for some Smart Phone models, there is be more than one *Add to cart* button.

When modeling, we therefore require two different classes, one for the Smart Phone model and one for each Smart Phone variation.

Listing 3.7: myshop/models/polymorphic/smartphone.py

```python
from shop.money import Money, MoneyMaker
from shop.money.fields import MoneyField
from .product import Product

class SmartPhoneModel(Product):
    """
    A generic smart phone model, which must be concretized by a model `SmartPhone` - see below.
    """
    BATTERY_TYPES = (
        (1, "Lithium Polymer (Li-Poly)"),
        (2, "Lithium Ion (Li-Ion)"),
    )
    WIFI_CONNECTIVITY = (
        (1, "802.11 b/g/n"),
    )
    BLUETOOTH_CONNECTIVITY = (
        (1, "Bluetooth 4.0"),
    )
    battery_type = models.PositiveSmallIntegerField(_("Battery type"),
        choices=BATTERY_TYPES)
    battery_capacity = models.PositiveIntegerField(_("Capacity"),
        help_text=_("Battery capacity in mAh"))
    ram_storage = models.PositiveIntegerField(_("RAM"),
        help_text=_("RAM storage in MB"))
    wifi_connectivity = models.PositiveIntegerField(_("WiFi"),
        choices=WIFI_CONNECTIVITY, help_text=_("WiFi Connectivity"))
    bluetooth = models.PositiveIntegerField(_("Bluetooth"),
        choices=BLUETOOTH_CONNECTIVITY,
        help_text=_("Bluetooth Connectivity"))
```

Here the method `get_price()` can only return the minimum, average or maximum price for our product. In this situation, most merchants extol the prices as: *Price starting at € 99.50*.

The concrete Smart Phone then is modeled as:

```
class SmartPhone(models.Model):
    product = models.ForeignKey(SmartPhoneModel,
        verbose_name=_("Smart-Phone Model"))
    product_code = models.CharField(_("Product code"),
        max_length=255, unique=True)
    unit_price = MoneyField(_("Unit price"), decimal_places=3,
        help_text=_("Net price for this product"))
    storage = models.PositiveIntegerField(_("Internal Storage"),
        help_text=_("Internal storage in MB"))

```

To proceed with purchasing, customers need some *Cart and Checkout* pages.

Model for a generic Commodity

For demo purposes, this polymorphic example adds another kind of Product model, a generic Commodity. Here instead of adding every possible attribute of our product to the model, we try to remain as generic as possible, and instead use a `PlaceholderField` as provided by **djangoCMS**.

This allows us to add any arbitrary information to our product's detail page. The only requirement for this to work is, that the rendering template adds a templatetag to render this placeholder.

Since the **djangoSHOP** framework looks in the folder `catalog` for a template named after its product class, adding this HTML snippet should do the job:

This detail template extends the default template of our site. Apart from the product's name (which has added as a convenience), this view remains empty when first viewed. In *Edit* mode, double clicking on the heading containing the product name, opens the detail editor for our commodity.

After switching into *Structure* mode, a placeholder named `Commodity Details` appears. Here we can add as many Cascade plugins as we want, by subdividing our placeholder into rows, columns, images, text blocks, etc. It allows us to edit the detail view of our commodity in whatever layout we like. The drawback using this approach is, that it can lead to inconsistent design and is much more labor intensive, than just editing the product's attributes together with their appropriate templates.

Configure the Placeholder

Since we use this placeholder inside a hard-coded Bootstrap column, we must provide a hint to Cascade about the widths of that column. This has to be done in the settings of the project:

This placeholder configuration emulates the Bootstrap column as declared by `<div class="col-xs-12">`.

3.6 Catalog Views

Now that we know how to create product models and how to administer them, lets have a look on how to route them to our views.

When editing the CMS page used for the products list view, open **Advanced Settings** and chose **Products List** from the select box labeled **Application**.

Then chose a template with at least one placeholder. Click onto **View on site** to change into front-end editing mode. Locate the main placeholder and add a **Row** followed by a **Column** plugin from the section **Bootstrap**. Below that column add a **Catalog List Views** plugin from section **Shop**. Then publish the page, it should not display any products yet.

3.6.1 Add products to the category

Open the detail view of a product in Django's administration backend. Locate the many-to-many select box labeled **Categories > Cms pages**. Select the pages where each product shall appear on.

On reloading the list view, the assigned products now shall be visible. Assure that they have been set to be active, otherwise they won't show up.

If you nest categories, products assigned to children will be also be visible on their parents pages.

Product Model Serializers

We already learned how to write model classes and model managers, so what are serializers for?

In **djangoSHOP** the response views do not distinguish whether the product's information shall be rendered as HTML or transferred via JSON. This gives us the ability to use the same business logic for web browsers rendering static HTML, single page web applications communicating via AJAX or native shopping applications for your mobile devices. This btw. is one of the great benefits when working with RESTful API's and thanks to the djangorestframework we don't even have to write any Django Views anymore.

For instance, try to open the list- or the detail view of any of the products available in the shop. Then in the browsers URL input field append `?format=api` or `?format=json` to the URL. This will render the pure product information, but without embedding it into HTML.

The REST API view is very handy while developing. If you want to hide this on your production system , then in your settingy.py remove `'rest_framework.renderers.BrowsableAPIRenderer'` from `REST_FRAMEWORK['DEFAULT_RENDERER_CLASSES']`.

In the shop's catalog, we need some functionality to render a list view for all products and we need a detail view to render each product type. The **djangoSHOP** framework supplies two such serializers:

3.6.2 Serialize for the Products List View

For each product we want to display in a list view, we need a serializer which converts the content of the most important fields of a product. Normally these are the Id, the name and price, the URL onto the detail view, a short description and a sample image.

The **djangoSHOP** framework does not know which of those fields have to be serialized, therefore it requires some help from the programmer:

Listing 3.8: myshop/product_serializers.py

```python
from shop.rest.serializers import ProductSummarySerializerBase
from myshop.models.polymorphic.product import Product

class ProductSummarySerializer(ProductSummarySerializerBase):
    class Meta:
        model = Product
        fields = ('id', 'product_name', 'product_url',
            'product_type', 'product_model', 'price')
```

All these fields can be extracted directly from the product model with the exception of the sample image. This is because we yet do not know the final dimensions of the image inside its HTML element such as `<img src="...">`, and we certainly want to resize it using PIL/Pillow before it is delivered. An easy way to solve this problem is to use the `SerializerMethodField`. Simply extend the above class to:

```python
from rest_framework.serializers import SerializerMethodField

class ProductSummarySerializer(ProductSummarySerializerBase):
    media = SerializerMethodField()

    def get_media(self, product):
    return self.render_html(product, 'media')
```

As you might expect, `render_html` assigns a HTML snippet to the field `media` in the serialized representation of our product. This method uses a template to render the HTML. The name of this template is constructed using the following rules:

- Look for a folder named according to the project's name, ie. `settings.SHOP_APP_LABEL` in lower case. If no such folder can be found, then use the folder named `shop`.
- Search for a subfolder named `products`.
- Search for a template named "*label-product_type-postfix*.html". These three subfieds are determined using the following rule: * *label*: the component of the shop, for instance `catalog`, `cart`, `order`. * *product_type*: the class name in lower case of the product's Django model, for instance `smartcard`, `smartphone` or if no such template can be found, just `product`. * *postfix*: This is an arbitrary name passed in by the rendering function. As in the example above, this is the string `media`.

Note: It might seem "un-restful" to render HTML snippets by a REST serializer and deliver them via JSON to the client. However, we somehow must re-size the images assigned to our product to fit into the layout of our list view. The easiest way to do this in a configurable manner is to use the easythumbnails library and its templatetag `{% thumbnail product.sample_image ... %}`.

The template to render the media snippet could look like:

Listing 3.9: myshop/products/catalog-smartcard-media.html

```
{% load i18n thumbnail djng_tags %}
{% thumbnail product.sample_image 100x100 crop as thumb %}
<img src="{{ thumb.url }}" width="{{ thumb.width }}" height="{{ thumb.height }}">
```

The template of the products list view then may contain a list iteration such as:

```
{% for product in data.results %}
  <div class="shop-list-item">
    <a href="{{ product.product_url }}">
      <h4>{{ product.product_name }}</h4>
        {{ product.media }}
        <strong>{% trans "Price" %}: {{ product.price }}</strong>
    </a>
  </div>
{% endfor %}
```

The tag `{{ product.media }}` inserts the HTML snippet as prepared by the serializer from above. A serializer may add more than one `SerializerMethodField`. This can be useful, if the list view shall render different product types using different snippet templates.

3.6.3 Serialize for the Product's Detail View

The serializer for the Product's Detail View is very similar to its List View serializer. In the example as shown below, we even reverse the field listing by explicitly excluding the fields we're not interested in, rather than naming the fields we want to include. This for the product's detail view makes sense, since we want to expose every possible detail.

```
from shop.rest.serializers import ProductDetailSerializerBase

class ProductDetailSerializer(ProductDetailSerializerBase):
    class Meta:
        model = Product
        exclude = ('active',)
```

3.6.4 The `AddToCartSerializer`

Rather than using the detail serializer, the business logic for adding a product to the cart has been moved into a specialized serializer. This is because **djangoSHOP** can not presuppose that products are added to the cart only from within the detail view[#add2cart]_. We also need a way to add more than one product variant to the cart from each products detail page.

For this purpose **djangoSHOP** is shipped with an `AddToCartSerializer`. It can be overridden for special product requirements, but for a standard application it just should work out of the box.

Assure that the context for rendering a product contains the key `product` referring to the product object. The `ProductDetailSerializer` does this by default. Then add

```
{% include "shop/catalog/product-add2cart.html" %}
```

to an appropriate location in the template which renders the product detail view.

The now included add-to-cart template contains a form with some input fields and a few AngularJS directives, which communicate with the endpoint connected to the `AddToCartSerializer`. It updates the subtotal whenever the customer changes the quantity and displays a nice popup window, whenever an item is added to the cart. Of course, that template can be extended with arbitrary HTML.

These Angular JS directives require some JavaScript code which is located in the file `shop/js/catalog.js`; it is referenced automatically when using the above template include statement.

Connect the Serializers with the View classes

Now that we declared the serializers for the product's list- and detail view, the final step is to access them through a CMS page. Remember, since we've chosen to use CMS pages as categories, we had to set a special **djangoCMS** apphook:

Listing 3.10: myshop/cms_app.py

```python
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool

class ProductsListApp(CMSApp):
    name = _("Products List")
    urls = ['myshop.urls.products']

apphook_pool.register(ProductsListApp)
```

This apphook points onto a list of boilerplate code containing these urlpattern:

Listing 3.11: myshop/urls/products.py

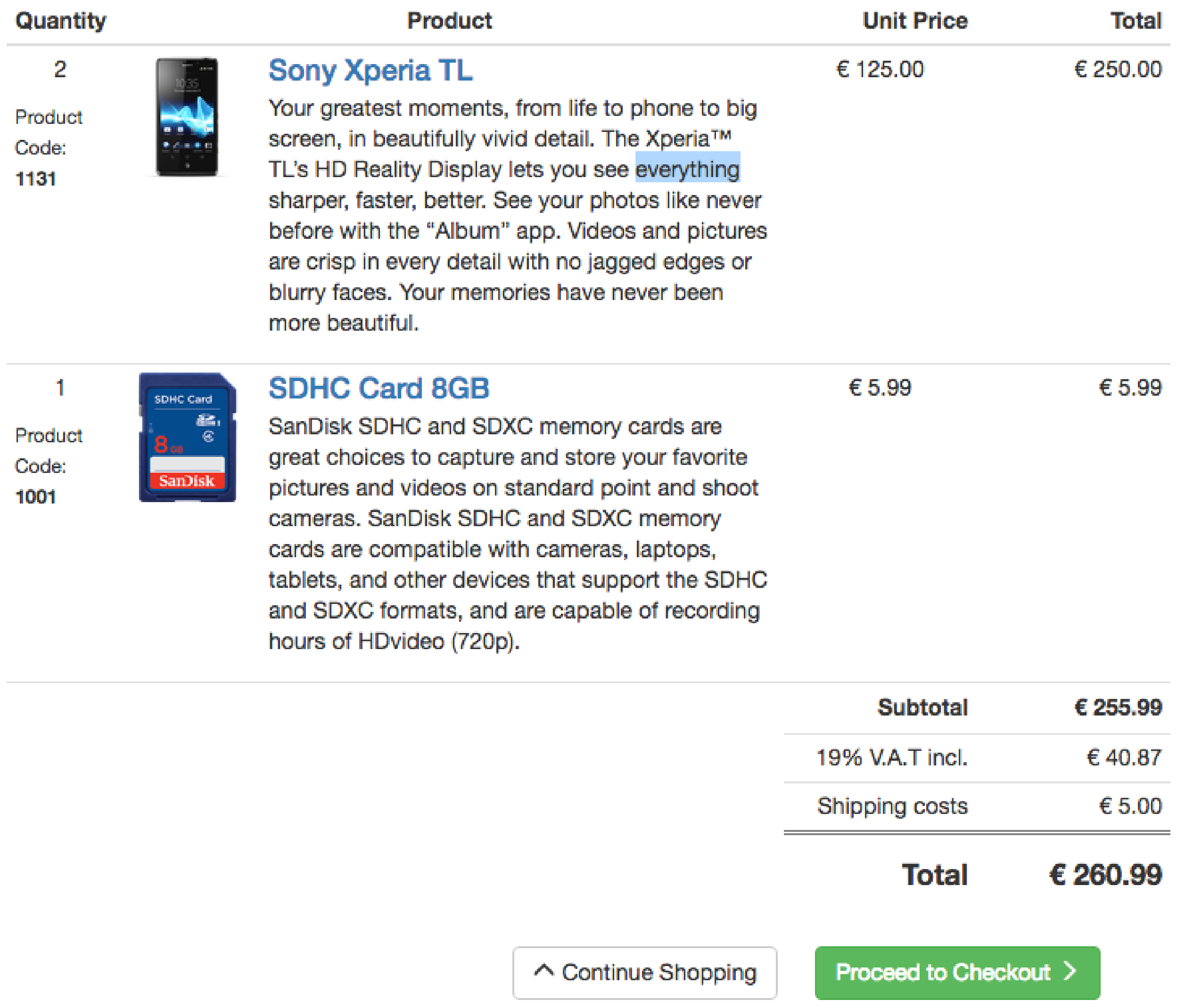
```python
from django.conf.urls import patterns, url
from rest_framework.settings import api_settings
from shop.rest.filters import CMSPagesFilterBackend
from shop.rest.serializers import AddToCartSerializer
from shop.views.catalog import (CMSPageProductListView,
    ProductRetrieveView, AddToCartView)

urlpatterns = patterns('',
    url(r'^$', CMSPageProductListView.as_view(
        serializer_class=ProductSummarySerializer,
    )),
    url(r'^(?P<slug>[\w-]+)$', ProductRetrieveView.as_view(
        serializer_class=ProductDetailSerializer
    )),
    url(r'^(?P<slug>[\w-]+)/add-to-cart', AddToCartView.as_view()),
)
```

These URL patterns connect the product serializers with the catalog views in order to assign them an endpoint. Additional note: The filter class `CMSPagesFilterBackend` is used to restrict products to specific CMS pages, hence it can be regarded as the product categoriser.

3.7 Cart and Checkout

In **djangoSHOP**, the cart and checkout view follow the same idea as all other pages – they are managed by the CMS. Change into the Django admin backend and look for the CMS page tree. A good position for adding a page is the root level, but then assure that in Advanced Setting the checkbox **Soft root** is set.

The checkout my or be combined with the cart on the same page, or moved on a separate page. Its best position normally is just below the Cart page.



Quantity		Product	Unit Price	Total
2 Product Code: **1131**		**Sony Xperia TL** Your greatest moments, from life to phone to big screen, in beautifully vivid detail. The Xperia™ TL's HD Reality Display lets you see everything sharper, faster, better. See your photos like never before with the "Album" app. Videos and pictures are crisp in every detail with no jagged edges or blurry faces. Your memories have never been more beautiful.	€ 125.00	€ 250.00
1 Product Code: **1001**		**SDHC Card 8GB** SanDisk SDHC and SDXC memory cards are great choices to capture and store your favorite pictures and videos on standard point and shoot cameras. SanDisk SDHC and SDXC memory cards are compatible with cameras, laptops, tablets, and other devices that support the SDHC and SDXC formats, and are capable of recording hours of HDvideo (720p).	€ 5.99	€ 5.99

Subtotal	**€ 255.99**
19% V.A.T incl.	€ 40.87
Shipping costs	€ 5.00
Total	**€ 260.99**

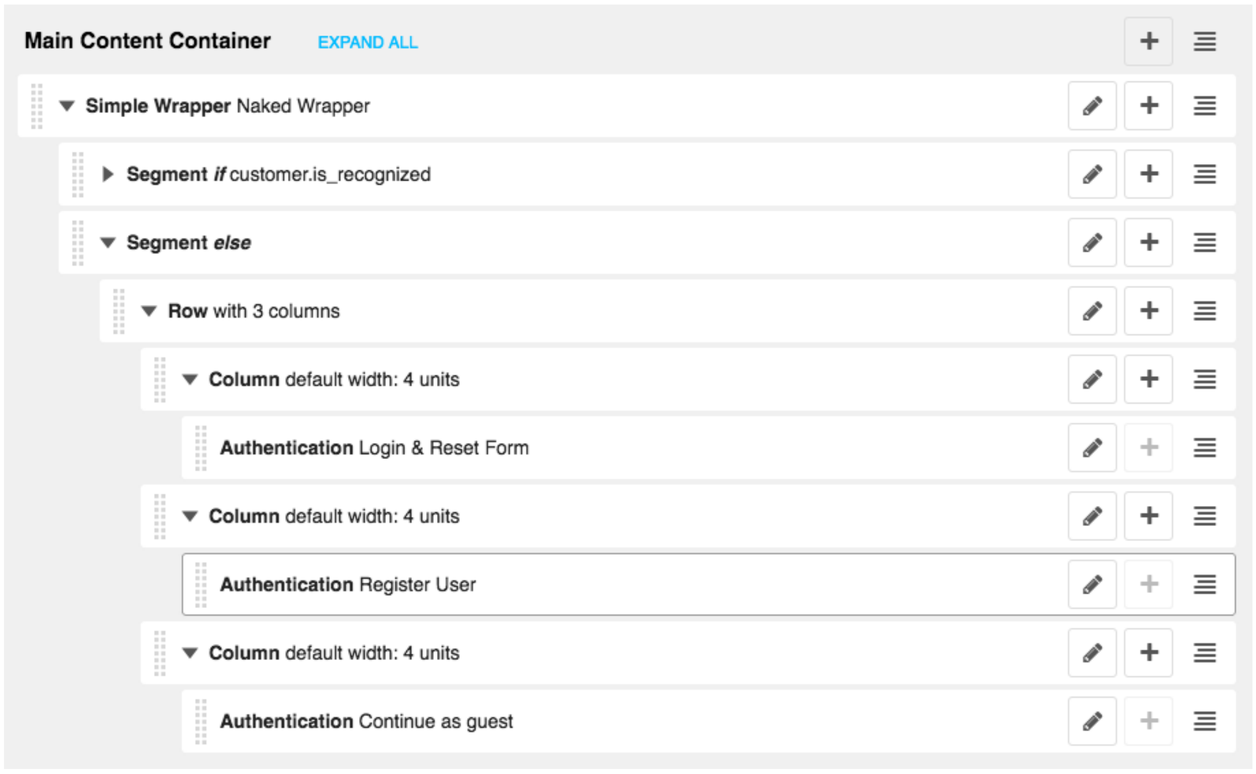
The Checkout pages presumably are the most complicated page to setup. Therefore no generic receipt can be presented here. Instead some CMS plugins will be listed here. They can be useful to compose a complete checkout page. In the reference section it is shown in detail how to create a *Cart and Checkout* view, but for this tutorial the best way to proceed is to have a look in the prepared demo project for the *Cart* and *Checkout* pages.

A list of plugins specific to **djangoSHOP** can be found in the reference section. They include a cart editor, a static cart renderer, forms to enter the customers names, addresses, payment- and shipping methods, credit card numbers and some more.

Other useful plugins can be found in the Django application djangocms-cascade.

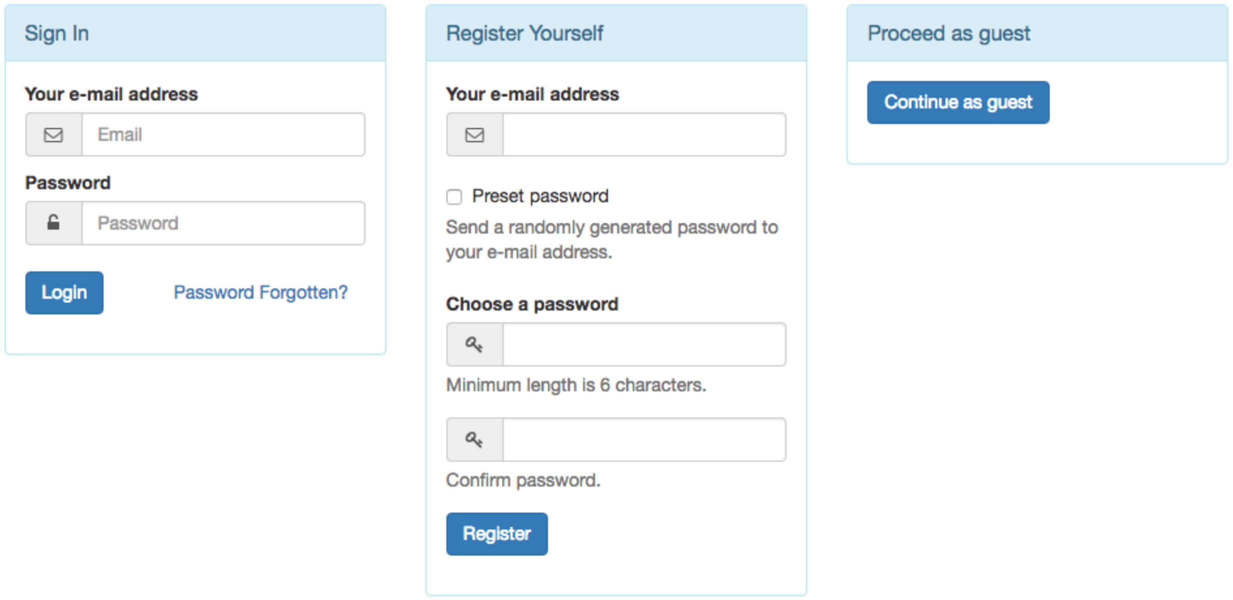
3.7.1 Scaffolding

Depending on who is allowed to buy products, keep in mind that pure visiting customers must declare themselves, whether they want to buy as guests or as registered users. This means that we first must distinguish between pure visitors and recognized customers. The simplest way to do this is to use the Segmentation **if**- and **else**-plugins. A recognized customer shall be able to proceed directly to the purchasing page. A visiting customer first must declare himself, this could be handled with a collections of plugins, such as:


Main Content Container EXPAND ALL

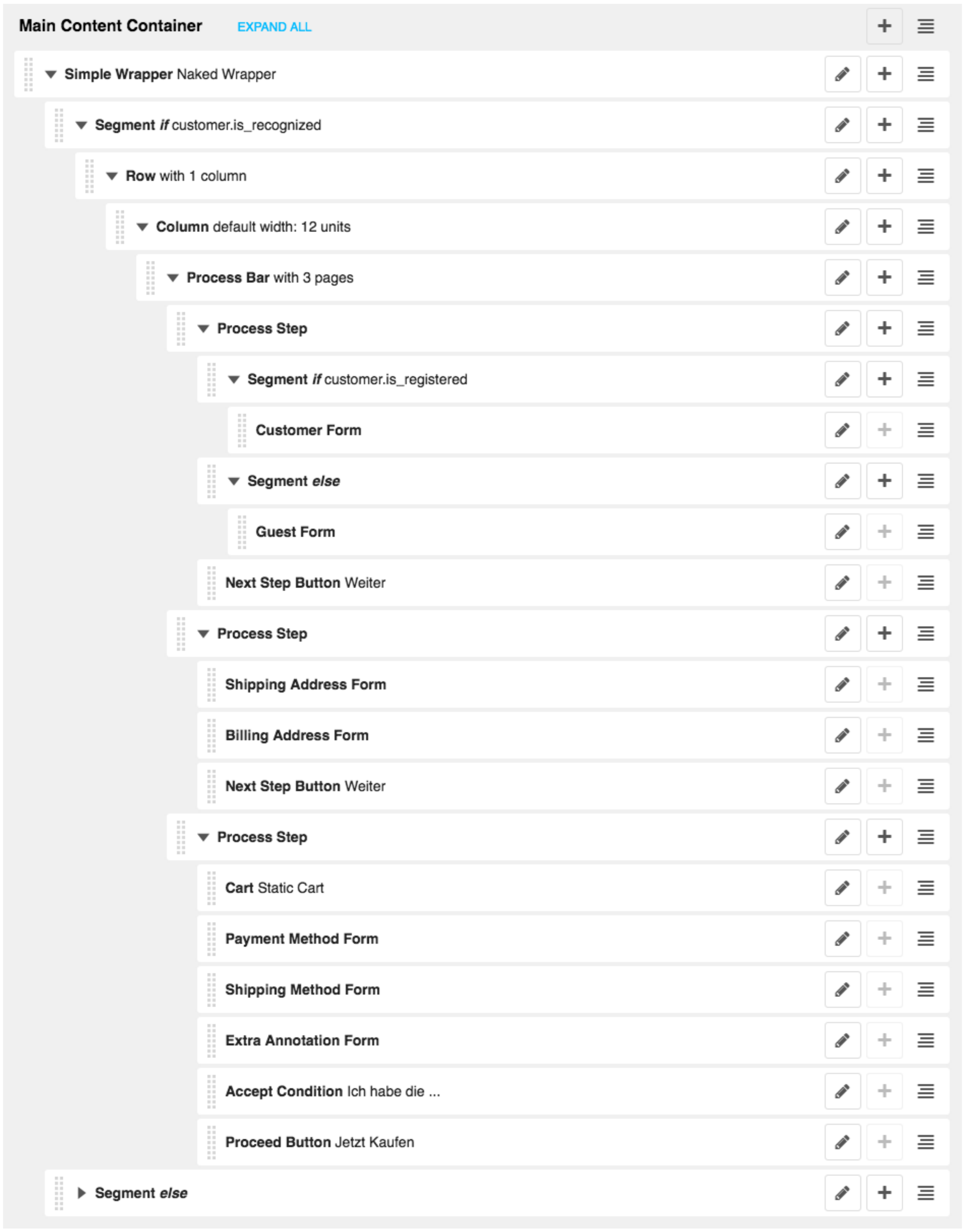
- **Simple Wrapper** Naked Wrapper
  - **Segment** *if* customer.is_recognized
  - **Segment** *else*
    - **Row** with 3 columns
      - **Column** default width: 4 units
        - **Authentication** Login & Reset Form
      - **Column** default width: 4 units
        - **Authentication** Register User
      - **Column** default width: 4 units
        - **Authentication** Continue as guest

in structure mode. This collection of plugins then will be rendered as:

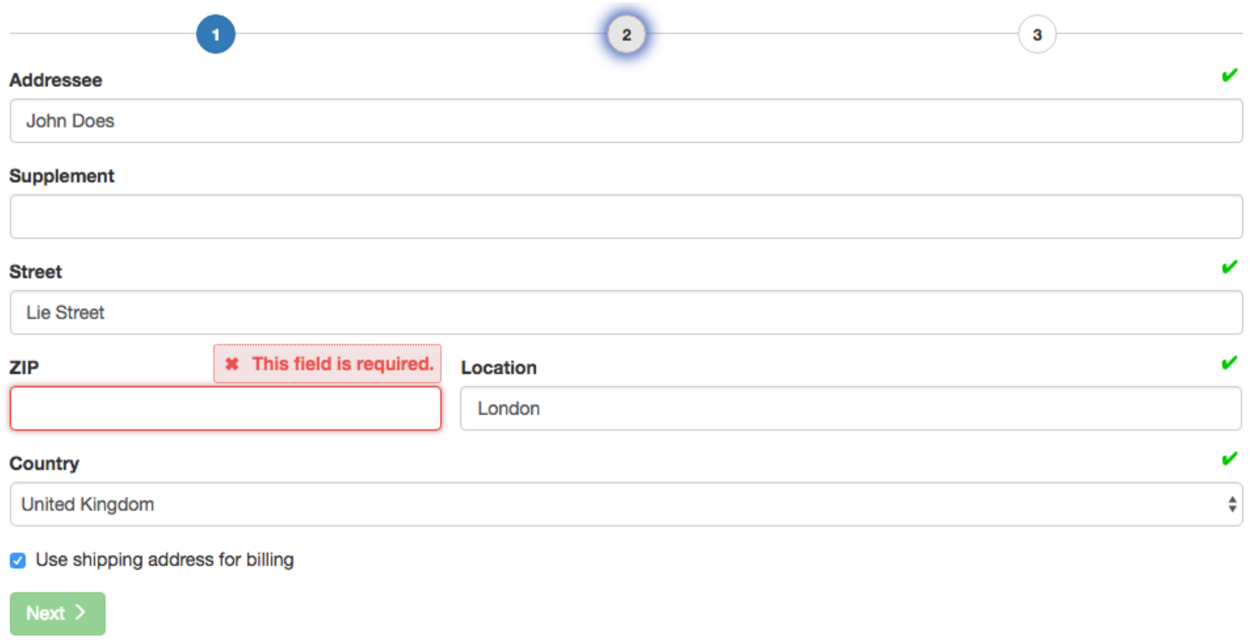


Please note that the Authentication plugins **Login & Reset**, **Register User** and **Continue as guest** must reload the current page. This is because during these steps a new session-id is assigned, which is requires a full page reload.

After reloading the page, the customer is considered as "recognized". Since there are a few forms to be filled, this example uses a **Process Bar** plugin, which emulates a few sub-pages, which then can be filled out by the customer step-by-step.


Main Content Container EXPAND ALL
Simple Wrapper Naked Wrapper
Segment *if* customer.is_recognized
Row with 1 column
Column default width: 12 units
Process Bar with 3 pages
Process Step
Segment *if* customer.is_registered
Customer Form
Segment *else*
Guest Form
Next Step Button Weiter
Process Step
Shipping Address Form
Billing Address Form
Next Step Button Weiter
Process Step
Cart Static Cart
Payment Method Form
Shipping Method Form
Extra Annotation Form
Accept Condition Ich habe die ...
Proceed Button Jetzt Kaufen
Segment *else*

A fragment of this collection of plugins then will be rendered as:



Addressee ✔

John Does

Supplement

Street ✔

Lie Street

ZIP ✖ This field is required.

Location ✔

London

Country ✔

United Kingdom

☑ Use shipping address for billing

CHAPTER 4

Reference

Reference to classes and concepts used in **djangoSHOP**

4.1 Customer Model

Most web applications distinguish logged in users explicitly from *the* anonymous site visitor, which is regarded as a non-existing user, and thus does not reference a session- or database entity. The Django framework, in this respect, is no exception.

This pattern is fine for web-sites, which run a Content Management System or a Blog, where only an elected group of staff users shall be permitted to access. This approach also works for web-services, such as social networks or Intranet applications, where visitors have to authenticate right from the beginning.

But when running an e-commerce site, this use-pattern has serious drawbacks. Normally, a visitor starts to look for interesting products, hopefully adding a few of them to their cart. Then on the way to the checkout, they decide whether to create a user account, use an existing one or continue as guest. Here's where things get complicated.

First of all, for non-authenticated site visitors, the cart does not belong to anybody. But each cart must be associated with its site visitor, hence the generic anonymous user object is not appropriate for this purpose. Unfortunately the Django framework does not offer an explicit but anonymous user object based on its session-Id.

Secondly, at the latest when the cart is converted into an order, but the visitor wants to continue as guest (thus remaining anonymous), that order object *must* refer to a User object in the database. These kind of users would be regarded as fakes, unable to log in, reset their password, etc. The only information which must be stored for such a faked User, is their email address otherwise they couldn't be informed, whenever the state of their order changes.

Django does not explicitly allow such User objects in its database models. But by using the boolean flag `is_active`, we can fool an application to interpret such *guest visitors* as a faked anonymous users.

However, since such an approach is unportable across all Django based applications, **djangoSHOP** introduces a new database model – the `Customer` model, which extends the existing `User` model.

4.1.1 Properties of the Customer Model

The `Customer` model has a 1:1 relation to the existing `User` model, which means that for each customer, there always exists *one and only one* user object. This approach allows us to do a few things:

The built-in User model can be swapped out and replaced against another implementation. Such an alternative implementation has a small limitation. It must inherit from `django.contrib.auth.models.AbstractBaseUser` and from `django.contrib.auth.models.PermissionMixin`. It also must define all the fields which are available in the default model as found in `django.contrib.auth.models.User`.

By setting the flag `is_active = False`, we can create guests inside Django's `User` model. Guests can not sign, they can not reset their password, and can thus be considered as "materialized" anonymous users.

Having guests with an entry in the database, gives us another advantage: By using the session key of the site visitor as the User object's `username`, it is possible to establish a link between a User object in the database with an otherwise anonymous visitor. This further allows the Cart and the Order models always refer to the User model, since they don't have to care about whether a certain User authenticated himself or not. It also keeps the workflow simple, whenever an anonymous User decides to register and authenticate himself in the future.

4.1.2 Adding the Customer model to our application

As almost all models in **djangoSHOP**, the Customer model uses itself the *Deferred Model Pattern*. This means that the Django project is responsible for materializing that model and additionally allows the merchant to add arbitrary fields to this Customer model. Good choices are a phone number, a boolean to signal whether the customer shall receive newsletters, his rebate status, etc.

The simplest way is to materialize the given convenience class in our project's `models.py`:

```
from shop.models.defaults.customer import Customer
```

or, if we need extra fields, then instead of the above, we write:

```
from shop.models.customer import BaseCustomer

class (BaseCustomer):
    birth_date = models.DateField("Date of Birth")
    # other customer related fields
```

Configure the Middleware

A Customer object is created automatically with each visitor accessing the site. Whenever Django's internal AuthenticationMiddleware adds an `AnonymousUser` to the request object, djangoSHOP's CustomerMiddleware adds a `VisitingCustomer` to the request object as well. Neither the `AnonymousUser` nor the `VisitingCustomer` are stored inside the database.

Whenever the AuthenticationMiddleware adds an instantiated `User` to the request object, djangoSHOP's CustomerMiddleware adds an instantiated `Customer` to the request object as well. If no associated `Customer` exists yet, the CustomerMiddleware creates one.

Therefore add the CustomerMiddleware *after* the AuthenticationMiddleware in the project's `settings.py`:

```
MIDDLEWARE_CLASSES = (
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'shop.middleware.CustomerMiddleware',
    ...
)
```

Configure the Context Processors

Additionally, some templates may need to access the customer object through the `RequestContext`. Therefore, add this context processor to the `settings.py` of the project.

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'shop.context_processors.customer',
    ...
)
```

Implementation Details

The Customer model has a non-nullable one-to-one relation to the User model. Hence each Customer is associated with exactly one User. For instance, accessing the hashed password can be achieved through `customer.user.password`. Some common fields and methods from the User model, such as `first_name`, `last_name`, `email`, `is_anonymous()` and `is_authenticated()` are accessible directly, when working with a Customer object. Saving an instance of type Customer also invokes the `save()` method from the associated User model.

The other direction – accessing the Customer model from a User – does not always work. Accessing an attribute that way fails if the corresponding Customer object is missing, ie. if there is no reverse relation from a Customer pointing onto the given User object.

```
>>> from django.contrib.auth import get_user_model
>>> user = get_user_model().create(username='bobo')
>>> print user.customer.salutation
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "django/db/models/fields/related.py", line 206, in __get__
    self.related.get_accessor_name()))
DoesNotExist: User has no customer.
```

This can happen for User objects added manually or by other applications.

During database queries, **djangoSHOP** always performs and INNER JOIN between the Customer and the User table. Therefore it performs better to query the User via the Customer object, rather than vice versa.

Anonymous Users and Visiting Customers

Most requests to our site will be of anonymous nature. They will not send a cookie containing a session-Id to the client, and the server will not allocate a session bucket. The middleware adds a `VisitingCustomer` object associated with an `AnonymousUser` object to the request. These two objects are not stored inside the database.

Whenever such an anonymous user/visiting customer adds the first item to the cart, **djangoSHOP** instantiates a User object in the database and associates it with a Customer object. Such a Customer is considered as "unregistered" and invoking `customer.is_authenticated()` will return False; its associated User model is inactive and has an unusable password.

Guests and Registered Customers

On the way to the checkout, a customer must declare himself, whether to continue as guest, to sign in using an existing account or to register himself with a new account. In the former case (customer wishes to proceed as guest), the User object remains as it is: Inactive and with an unusable password. In the second case, the visitor signs in using Django's default authentication backends. Here the cart's content is merged with the already existing cart of that user object. In the latter case (customer registers himself), the user object is recycled and becomes an active Django User object, with a password and an email address.

Obviate Criticism

Some may argue that adding unregistered and guest customers to the User table is an anti-pattern or hack. So, what are the alternatives?

We could keep the cart of anonymous customers in the session store. This was the procedure used until **djangoSHOP** version 0.2. It however required to keep two different models of the cart, one session based and one relational. Not very practical, specially if the cart model should be overridable by the merchant's own implementation.

We could associate each cart models with a session id. This would require an additional field which would be NULL for authenticated customers. While possible in theory, it would require a lot of code which distinguishes between anonymous and authenticated customers. Since the aim of this software is to remain simple, this idea was dismissed.

We could keep the primary key of each cart in the session associated with the customer. But this would it make very hard to find expired carts, because we would have to iterate over all carts and for each cart we would have to iterate over all sessions to check if the primary keys matches. Remember, there is no such thing as an OUTER JOIN between sessions and database tables.

We could create a customer object which is independent of the user. Hence instead of having a `OneToOneField(AUTH_USER_MODEL)` in model `Customer`, we'd have this 1:1 relation with a nullable foreign key. This would require an additional field to store the session id in the customer model. It also would require an additional email field, if we wanted a guest customers to remain anonymous users – what they actually are, since they can't sign in. Apart from field duplication, this approach would also require some code to distinguish between unrecognized, guest and registered customers. In addition to that, the administration backend would require two distinguished views, one for the customer model and one for the user model.

4.1.3 Authenticating against the Email Address

Nowadays it is quite common, to use the email address for authenticating, rather than an explicit account identifier. This in Django is not possible without replacing the built-in User model. Since for an e-commerce site this authentication variant is rather important, **djangoSHOP** is shipped with an optional drop-in replacement for the built-in User model.

This convenience User model is almost a copy of the existing `User` model as found in `django.contrib.auth.models.py`, but it uses the field `email` rather than `username` for looking up the credentials. To activate this alternative User model, add to the project's `settings.py`:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'email_auth',
    ...
)

AUTH_USER_MODEL = 'email_auth.User'
```

Note: This alternative User model uses the same database table as the Django authentication would, namely `auth_user`. It is even field-compatible with the built-in model and hence can be added later to an existing Django project.

Caveat when using this alternative User model

The savvy reader may have noticed that in `email_auth.models.User`, the email field is not declared as unique. This by the way causes Django to complain during startup with:

```
WARNINGS:
email_auth.User: (auth.W004) 'User.email' is named as the 'USERNAME_FIELD', but it is not unique.
    HINT: Ensure that your authentication backend(s) can handle non-unique usernames.
```

This warning can be silenced by adding `SILENCED_SYSTEM_CHECKS = ['auth.W004']` to the project's `settings.py`.

The reason for this is twofold:

First, Django's default user model has no unique constraint on the email field, so `email_auth` remains more compatible.

Second, the uniqueness is only required for users which actually can sign in. Guest users on the other hand can not sign in, but they may return someday. By having a unique email field, the Django application `email_auth` would lock them out and guests would be allowed to buy only once, but not a second time – something we certainly do not want!

Therefore **djangoSHOP** offers two configurable options:

- Customers can declare herself as guests, each time they buy something. This is the default, but causes to have non-unique email addresses in the database.
- Customer can declare themselves as guests the first time they buys something. If someday they buy again, they will be recognized as returning customer and must use a form to reset their password. This configuration can be activated with `SHOP_GUEST_IS_ACTIVE_USER = True` in the project's `settings.py`. This allows us, to set a unique constraint on the email field.

Note: The email field from Django's built-in User model has a max-length of 75 characters. This is enough for most use-cases but violates RFC-5321, which requires 254 characters. The alternative implementation uses the correct max-length.

Administration of Users and Customers

By keeping the Customer and the User model tight together, it is possible to reuse the Django's administration backend for both of them. All we have to do is to import and register the Customer backend inside the project's `admin.py`:

```
from django.contrib import admin
from shop.admin.customer import CustomerProxy, CustomerAdmin

admin.site.register(CustomerProxy, CustomerAdmin)
```

This administration backend recycles the built-in `django.contrib.auth.admin.UserAdmin`, and enriches it by adding the Customer model as a `StackedInlineAdmin` on top of the detail page. By doing so, we can edit the Customer and User fields on the same page.

4.1.4 Summary for Customer to User mapping

This table summarizes to possible mappings between a Django User Model [1] and the Shop's Customer model:

[1] or any alternative User model, as set by `AUTH_USER_MODEL`.

Shop's Customer Model	Django's User Model	Active Session
VisitingCustomer object	AnonymousUser object	No
Unrecognized Customer	Inactive User object with unusable password	Yes, but not logged in
Customer recognized as guest [2]	Inactive User with valid email address but unusable password	Yes, but not logged in
Customer recognized as guest [3]	Active User with valid email address and unknown, but resetable password	Yes, but not logged in
Registered Customer	Active User with valid email address, known password, optional salutation, first- and last names	Yes, logged in using Django's authentication backend

Manage Customers

djangoSHOP is shipped with a special management command which informs the merchant about the state of customers. In the project's folder, invoke on the command line:

```
./manage.py shop_customers
Customers in this shop: total=20482, anonymous=17418, expired=10111, active=1068, guests=1997, regist
```

Read these numbers as: * Anonymous customers are those which added at least one item to the cart, but never proceeded to checkout. * Expired customers are the subset of the anonymous customers, whose session already expired. * The difference between guest and registered customers is explained in the above table.

Delete expired customers

By invoking on the command line:

```
./manage.py shop_customers --delete-expired
```

This removes all anonymous/unregistered customers and their associated user entities from the database, whose session expired. This command may be used to reduce the database storage requirements.

4.2 Deferred Model Pattern

Until **djangoSHOP** version 0.2, there were abstract and concrete and models: BaseProduct and Product, BaseCart and Cart, BaseCartItem and CartItem, BaseOrder and Order and finally, BaseOrderItem and OrderItem.

The concrete models were stored in shop.models, whereas abstract models were stored in shop.models_bases. This was quite confusing and made it difficult to find the right model definition whenever one had to access the definition of one of the models. Additionally, if someone wanted to subclass a model, he had to use a configuration directive, say PRODUCT_MODEL, ORDER_MODEL, ORDER_MODEL_ITEM from the projects settings.py.

This made configuration quite complicate and causes other drawbacks:

[2] if setting SHOP_GUEST_IS_ACTIVE_USER = False (the default).
[3] if setting SHOP_GUEST_IS_ACTIVE_USER = True.

- Unless *all* models have been overridden, the native ones appeared in the administration backend below the category *Shop*, while the customized ones appeared under the given project's name. To merchants, this inconsistency in the backend was quite difficult to explain.
- In the past, mixing subclassed with native models caused many issues with circular dependencies.

Therefore in **djangoSHOP**, since version 0.9 *all* concrete models, `Product`, `Order`, `OrderItem`, `Cart`, `CartItem` have been removed. These model definitions now all are abstract and named `BaseProduct`, `BaseOrder`, `BaseOrderItem`, etc. They all have been moved into the folder `shop/models/`, because that's the location a programmer expects them.

4.2.1 Materializing Models

Materializing such an abstract base model, means to create a concrete model with an associated database table. This model creation is performed in the concrete project implementing the shop; it must be done for each base model in the shop software.

For instance, materialize the cart by using this code snippet inside our own shop's `models/shopmodels.py` files:

```
from shop.models import cart

class Cart(cart.BaseCart):
    my_extra_field = ...

    class Meta:
        app_label = 'my_shop'

class CartItem(cart.BaseCartItem):
    other_field = ...

    class Meta:
        app_label = 'my_shop'
```

Of course, we can add as many extra model fields to this concrete cart model, as we wish. All shop models, now are managed through *our* project instance. This means that the models **Cart**, **Order**, etc. are now managed by the common database migrations tools, such as `./manage.py makemigration my_shop` and `./manage.py migrate my_shop`. This also means that these models, in the Django admin backend, are visible under **my_shop**.

Use the default Models

Often we don't need extra fields, hence the abstract shop base model is enough. Then, materializing the models can be done using some convenience classes as found in `shop/models/defaults`. We can simply import them into `models.py` or `models/__init__.py` in our own shop project:

```
from shop.models.defaults.cart import Cart  # nopyflakes
from shop.models.defaults.cart_item import CartItem  # nopyflakes
```

Note: The comment `nopyflakes` has been added to suppress warnings, since these classes arern't used anywhere in `models.py`.

All the configuration settings from **djangoSHOP** <0.9: `PRODUCT_MODEL`, `ORDER_MODEL`, `ORDER_MODEL_ITEM`, etc. are not required anymore and can safely be removed from our `settings.py`.

4.2.2 Accessing the deferred models

Since models in **djangoSHOP** are yet unknown during instantiation, one has to access their materialized instance using the lazy object pattern. For instance in order to access the Cart, use:

```
from shop.models.cart import CartModel

def my_view(request):
    cart = CartModel.objects.get_from_request(request)
    cart.items.all()  # contains the queryset for all items in the cart
```

Here `CartModel` is a lazy object resolved during runtime and pointing on the materialized, or, to say it in other words, real Cart model.

4.2.3 Technical Internals

Mapping of Foreign Keys

One might argue, that this can’t work, since foreign keys must refer to a real model, not to abstract ones! Therefore one can not add a field `ForeignKey`, `OneToOneField` or `ManyToManyField` which refers an abstract model in the **djangoSHOP** project. But relations are fundamental for a properly working software. Imagine a `CartItem` without a foreign relation to `Cart`.

Fortunately there is a neat trick to solve this problem. By deferring the mapping onto a real model, instead of using a real `ForeignKey`, one can use a special “lazy” field, declaring a relation with an abstract model. Now, whenever the models are “materialized”, then these abstract relations are converted into real foreign keys. The only drawback for this solution is, that one may derive from an abstract model only once, but for **djangoSHOP** that’s a non-issue and doesn’t differ from the current situation, where one can subclass `BaseCart` only once anyway.

Therefore, when using this deferred model pattern, instead of using `models.ForeignKey`, `models.OneToOneField` or `models.ManyToManyField`, use the special fields `deferred.ForeignKey`, `deferred.OneToOneField` and `deferred.ManyToManyField`. When Django materializes the model, these deferred fields are resolved into real foreign keys.

Accessing the materialized model

While programming with abstract model classes, sometimes they must access their model manager or their concrete model definition. A query such as `BaseCartItem.objects.filter(cart=cart)` therefore can not function and will throw an exception. To facilitate this, the deferred model’s metaclasses adds an additional member `_materialized_model` to their base class, while building the model class. This model class then can be accessed through lazy evaluation, using `CartModel`, `CartItemModel`, `OrderModel`, `OrderItemModel`, etc.

4.3 Money Types

Until **djangoSHOP** version 0.2, amounts relating to money were kept inside a `Decimal` type and stored in the database model using a `DecimalField`. In shop installations with only one available currency, this wasn’t a major issue, because the currency symbol could be hard-coded anywhere on the site.

However, for sites offering pricing information in more than one currency, this caused major problems. When we needed to perform calculations with amounts that have an associated currency, it is very common to make mistakes by mixing different currencies. It also is common to perform incorrect conversions that generate wrong results. Python doesn’t allow developers to associate a specific decimal value with a unit.

Starting with version 0.3.0, **djangoSHOP** now is shipped with a special factory class:

4.3.1 MoneyMaker

This class can not be instantiated, but is a factory for building a money type with an associated currency. Internally it uses the well established `Decimal` type to keep track of the amount. Additionally, it restricts operations on the current Money type. For instance, we can't sum up Dollars with Euros. We also can't multiply two currencies with each other.

Not a Number

In special occurrences we'd rather want to specify "no amount" rather than an amount of 0.00 (zero). This can be useful for free samples, or when an item is currently not available. The Decimal type denotes a kind of special value a `NaN` – for "Not a Number". Our Money type also knows about this special value, and when rendered, `€ –` is printed out.

Declaring a Money object without a value, say `m = Money()` creates such a special value. The big difference as for the `Decimal` type is that when adding or subtracting a `NaN` to a valid value, it is considered zero, rather than changing the result of this operation to `NaN`.

It also allows us to multiply a Money amount with `None`. The result of this operation is `NaN`.

Create a Money type

```
>>> from shop.money_maker import MoneyMaker
>>> Money = MoneyMaker()
>>> print Money('1.99')
€ 1.99

>>> print Money('1.55') + Money('8')
€ 9.55

>>> print Money
<class 'shop.money.money_maker.MoneyInEUR'>

>>> Yen = MoneyMaker('JPY')
>>> print Yen('1234.5678')
¥ 1235

>>> print Money('100') + Yen('1000')
ValueError: Can not add/substract money in different currencies.
```

How does this work?

By calling `MoneyMaker()` a type accepting amounts in the *default currency* is created. The default currency can be changed in `settings.py` with `SHOP_DEFAULT_CURRENCY = 'USD'`, using one of the official ISO-4217 currency codes.

Alternatively, we can create our own Money type, for instance `Yen`.

Formating Money

When the amount of a money type is printed or forced to text using `str(price)`, it is prefixed by the currency symbol. This is fine, when working with only a few currencies. However, some symbols are ambiguous, for instance Canadian, Australian and US Dollars, which all use the "$" symbol.

With the setting `SHOP_MONEY_FORMAT` we can style how money is going to be printed out. This setting defaults to `{symbol} {amount}`. The following format strings are allowed:

- `{symbol}`: The short symbol for a currency, for instance `$`, `£`, `€`, `¥`, etc.
- `{code}`: The international currency code, for instance USD, GBP, EUR, JPY, etc.
- `{currency}`: The spoken currency description, for instance "US Dollar", "Pound Sterling", etc.
- `{amount}`: The amount, unlocalized.

Thus, if we prefer to print `9.98 US Dollar`, then we should set `{amount} {currency}` as the formatting string.

4.3.2 Localizing Money

Since the Money class doesn't know anything about our current locale setting, amounts always are printed unlocalized. To localize a Money type, use `django.utils.numberformat.format(someamount).` This function will return the amount, localized according to the current HTTP request.

4.3.3 Money Database Fields

Money can be stored in the database, keeping the currency information together with the field type. Internally, the database uses the Decimal type, but such a field knows its currency and will return an amount as `MoneyIn...` type. This prevents implicit, but accidental currency conversions.

In our database model, declare a field as:

```
class Product(models.Model):
    ...
    unit_price = MoneyField(currency='GBP')
```

This field stores its amounts as British Pounds and returns them typed as `MoneyInGBP`. If the `currency` argument is omitted, then the default currency is used.

4.3.4 Money Representation in JSON

An additional REST SerializerField has been added to convert amounts into JSON strings. When writing REST serializers, use:

```
from rest_framework import serializers
from shop.money.rest import MoneyField

class SomeSerializer(serializers.ModelSerializer):
    price = MoneyField()
```

The default REST behavior serializes Decimal types as floats. This is fine if we want to do some computations in the browser using JavaScript. However, then the currency information is lost and must be re-added somehow to the output strings. It also is a bad idea to do commercial calculations using floats, yet JavaScript does not offer any Decimal-like type. I therefore recommend to always do the commerce calculations on the server and transfer amount information using JSON strings.

4.4 Product Models

Products can vary wildly, and modeling them is not always trivial. Some products are salable in pieces, while others are continues. Trying to define a set of product models, capable for describing all such scenarios is impossible –

4.4.1 Describe Products by customizing the Model

DjangoSHOP requires to describe products instead of prescribing prefabricated models.

All in all, we know best how our products should be modelled!

E-commerce solutions, claiming to be plug-and-play, usually use one of these (anti-)patterns

Either, they offer a field for every possible variation, or they use the Entity Attribute Value (EAV) pattern to add meta-data for each of our models. This at a first glance seems to be easy. But both approaches are unwieldy and have serious drawbacks. They both apply a different "physical schema" – the way data is stored, rather than a "logical schema" – the way users and applications require that data. As soon as we have to combine our e-commerce solution with some Enterprise Resource Planning (ERP) software, additional back-and-forward conversion routines have to be added.

In djangoSHOP, the physical representation of a product always maps to its logical

djangoSHOP's approach to this problem is to have a minimal set of models. These abstract models are stubs providing to subclass the physical models. Hence the logical representation of the product conforms to their physical one. Moreover, it is even possible to represent various types of products by subclassing polymorphically from an abstract base model. Thanks to Django's Object Relational Mapper, modeling the logical representation for a set of products, together with an administration backend, becomes almost effortless.

Therefore the base class to model a product is a stub which contains only these three fields:

The timestamps for `created_at` and `updated_at`; these are self-explanatory.

A boolean field `active`, used to signalize the products availability.

The attentive reader may wonder, why there not even fields for the most basic requirements of each sellable article, there is no product name, no price field and no product code.

The reason for this is, that **djangoSHOP** does not impose any fields, which might require a different implementation for the merchants use case. However, for a sellable commodity some information is fundamental and required. But its up to him how to implement these fields:

The product's name must be implemented as a model field or as a property method, but both must be declared as `product_name`. Use a method implementation for composed and translatable names, otherwise use a database model field with that name.

The product's price must be implemented as a method declared as `get_price(request)` which accepts the request object. This gives the merchant the ability to vary the price and/or its currency depending on the geographic location, the customers login status, the browsers user-agent, or whatever else.

An optional, but highly recommended field is the products item number, declared as `product_code`. It shall return a unique and language independent identifier for each product, to be identifiable. In most cases the product code is implemented by the product model itself, but in some circumstances it may be implemented by the product's variant. The `SmartPhone` from the demo code is one such example.

The example section of **djangoSHOP** contains a few models which can be copied and adopted to the specific needs of the merchants products. Let's have a look at a few use-cases:

4.4.2 Case study: Smart-Phones

There are many smart-phone models with different equipment. All the features are the same, except for the built-in storage. How shall we describe such a model?

In that model, the product's name shall not be translatable, not even on a multi-lingual site, since smart-phones have international names used everywhere. Smart-phones models have dimensions, an operating system, a display type and other features.

But smart-phone have different equipment, namely the built-in storage, and depending on that, they have different prices and a unique product code. Therefore our product models consists of two classes, the generic smart phone model and the concrete flavor of that model.

Therefore we would model our smart-phones using a database model similar to the following one:

```
from shop.models.product import BaseProductManager, BaseProduct
from shop.money import Money

class SmartPhoneModel(BaseProduct):
    product_name = models.CharField(max_length=255,
        verbose_name=_("Product Name"))
    slug = models.SlugField(verbose_name=_("Slug"))
    description = HTMLField(help_text=_("Detailed description."))
    manufacturer = models.ForeignKey(Manufacturer,
        verbose_name=_("Manufacturer"))
    screen_size = models.DecimalField(_("Screen size"),
        max_digits=4, decimal_places=2)
    # other fields to map the specification sheet

    objects = BaseProductManager()
    lookup_fields = ('product_name__icontains',)

    def get_price(request):
        aggregate = self.smartphone_set.aggregate(models.Min('unit_price'))
        return Money(aggregate['unit_price__min'])

class SmartPhone(models.Model):
    product_model = models.ForeignKey(SmartPhoneModel)
    product_code = models.CharField(_("Product code"),
        max_length=255, unique=True)
    unit_price = MoneyField(_("Unit price"))
    storage = models.PositiveIntegerField(_("Internal Storage"))
```

Lets go into the details of these classes. The model fields are self-explanatory. Something to note here is, that each product requires a field `product_name`. This alternatively can also be implemented as property.

Another mandatory attribute for each product is the `ProductManager` class. It must inheriting from `BaseProductManager`, and adds some methods to generate some special querysets.

Finally, the attribute `lookup_fields` contains a list or tuple of lookup fields. These are required by the administration backend, and used when the site editor has to search for certain products. Since the framework does not impose which fields are used to distinguish between products, we must give some hints.

Each product also requires a method implemented as `get_price(request)`. This must return the unit price using one of the available *Money Types*.

Add multilingual support

Adding multilingual support to an existing product is quite easy and straight forward. To achieve this **djangoSHOP** uses the app django-parler which provides Django model translations without nasty hacks. All we have to do, is to replace the ProductManager with one capable of handling translations:

```
class ProductQuerySet(TranslatableQuerySet, PolymorphicQuerySet):
    pass
```

```
class ProductManager(BaseProductManager, TranslatableManager):
    queryset_class = ProductQuerySet
```

The next step is to locate the model fields, which shall be available in different languages. In our use-case thats only the product's description:

```
class SmartPhoneModel(BaseProduct, TranslatableModel):
    # other field remain unchanged
    description = TranslatedField()

class ProductTranslation(TranslatedFieldsModel):
    master = models.ForeignKey(SmartPhoneModel, related_name='translations', null=True)
    description = HTMLField(help_text=_("Some more detailed description."))

    class Meta:
        unique_together = [('language_code', 'master')]
```

This simple change now allows us to offer the shop's assortment in different natural languages.

Add Polymorphic Support

If besides smart phones we also want to sell cables, pipes or smart cards, we must split our product models into a common- and a specialized part. That said, we must separate the information every product requires from the information specific to a certain product type. Say, in addition to smart phones, we also want to sell smart cards. First we declare a generic `Product` model, which is a common base class of both, `SmartPhone` and `SmartCard`:

```
class Product(BaseProduct, TranslatableModel):
    product_name = models.CharField(max_length=255, verbose_name=_("Product Name"))
    slug = models.SlugField(verbose_name=_("Slug"), unique=True)
    description = TranslatedField()

    objects = ProductManager()
    lookup_fields = ('product_name__icontains',)
```

Next we only add the product specific attributes to the class models derived from `Product`:

```
class SmartPhoneModel(Product):
    manufacturer = models.ForeignKey(Manufacturer, verbose_name=_("Manufacturer"))
    screen_size = models.DecimalField(_("Screen size"), max_digits=4, decimal_places=2)
    battery_type = models.PositiveSmallIntegerField(_("Battery type"), choices=BATTERY_TYPES)
    battery_capacity = models.PositiveIntegerField(help_text=_("Battery capacity in mAh"))
    ram_storage = models.PositiveIntegerField(help_text=_("RAM storage in MB"))
    # and many more attributes as found on the data sheet

class SmartPhone(models.Model):
    product_model = models.ForeignKey(SmartPhoneModel)
    product_code = models.CharField(_("Product code"), max_length=255, unique=True)
    unit_price = MoneyField(_("Unit price"))
    storage = models.PositiveIntegerField(_("Internal Storage"))

class SmartCard(Product):
    product_code = models.CharField(_("Product code"), max_length=255, unique=True)
    storage = models.PositiveIntegerField(help_text=_("Storage capacity in GB"))
    unit_price = MoneyField(_("Unit price"))
    CARD_TYPE = (2 * ('{}{}'.format(s, t),) for t in ('SD', 'SDXC', 'SDHC', 'SDHC II') for s in ('',
    card_type = models.CharField(_("Card Type"), choices=CARD_TYPE, max_length=15)
    SPEED = ((str(s), "{} MB/s".format(s)) for s in (4, 20, 30, 40, 48, 80, 95, 280))
    speed = models.CharField(_("Transfer Speed"), choices=SPEED, max_length=8)
```

If *MyShop* would sell the iPhone5 with 16GB and 32GB storage as independent products, then we could unify the classes `SmartPhoneModel` and `SmartPhone` and move the attributes `product_code` and `unit_price` into the class `Product`. This would simplify some programming aspects, but would require the merchant to add a lot of information twice. Therefore we remain with the model layout presented here.

4.4.3 Caveat using a `ManyToManyField` with existing models

Sometimes we may need to use a `ManyToManyField` for models which are handled by other apps in our project. This for example could be an attribute `files` referring the model `filer.FilerFileField` from the library django-filer. Here Django would try to create a mapping table, where the foreign key to our product model can not be resolved properly, because while bootstrapping the application, our Product model is still considered to be deferred.

Therefore, we have to create our own mapping model and refer to it using the `through` parameter, as shown in this example:

```
from six import with_metaclass
from django.db import models
from filer.fields.file import FilerFileField
from shop.models import deferred
from shop.models.product import BaseProductManager, BaseProduct

class ProductFile(with_metaclass(deferred.ForeignKeyBuilder, models.Model)):
    file = FilerFileField()
    product = deferred.ForeignKey(BaseProduct)

class Product(BaseProduct):
    # other fields
    files = models.ManyToManyField('filer.File', through=ProductFile)

    objects = ProductManager()
```

Note: Do not use this example for creating a many-to-many field to `FilerImageField`. Instead use `shop.models.related.BaseProductImage` which is a base class for this kind of mapping. Just import and materialize it, in your own project.

4.5 Catalog

The catalog probably is that part, where customers of our e-commerce site spend the most time. Often it even makes sense, to start the *Catalog List View* on the main landing page.

In this documentation we presume that categories of products are built up using specially tagged CMS pages in combination with a djangoCMS apphook. This works perfectly well for most implementation, but some sites may require categories implemented independently of the CMS.

Using an external **djangoSHOP** plugin for managing categories is a very conceivable solution, and we will see separate implementations for this feature request. Using such an external category plugin can make sense, if this e-commerce site requires hundreds of hierarchical levels and/or these categories require a set of attributes which are not available in CMS pages. If you are going to use externally implemented categories, please refer to their documentation, since here we proceed using CMS pages as categories.

A nice aspect of **djangoSHOP** is, that it doesn't require the programmer to write any special Django Views in order to render the catalog. Instead all merchant dependent business logic goes into a serializer, which in this documentation is referred as `ProductSummarySerializer`.

4.5.1 Catalog List View

In this documentation, the catalog list view is implemented as a **djangoCMS** page. Depending on whether the e-commerce aspect of that site is the most prominent part, or just a niche of the CMS select an appropriate location in the page tree and create a new page. This will become the root of our catalog list.

But first we must *Create the ProductsListApp*.

Create the `ProductsListApp`

To retrieve a list of product models, the Catalog List View requires a djangoCMS apphook. This `ProductsListApp` must be added into a file named `cms_app.py` and located in the root folder of the merchant's project:

Listing 4.1: myshop/cms_app.py

```python
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool

class ProductsListApp(CMSApp):
    name = _("Catalog List")
    urls = ['myshop.urls.products']

apphook_pool.register(ProductsListApp)
```

as all apphooks, it requires a file defining its urlpatterns:

Listing 4.2: myshop/urls/products.py

```python
from django.conf.urls import patterns, url
from rest_framework.settings import api_settings
from shop.views.catalog import CMSPageProductListView
from myshop.serializers import ProductSummarySerializer

urlpatterns = patterns('',
    url(r'^$', CMSPageProductListView.as_view(
        serializer_class=ProductSummarySerializer,
    )),
    # other patterns
)
```

Here the `ProductSummarySerializer` serializes the product models into a representation suitable for being rendered inside a CMS page, as well as being converted to JSON. This allows us to reuse the same Django View (`CMSPageProductListView`) whenever the catalog list switches into infinite scroll mode, where it only requires the product's summary digested as JavaScript objects.

In case we need *Additional Product Serializer Fields*, lets add them to this class using the serializer fields from the Django RESTFramework library.

Add the Catalog to the CMS

In the page list editor of **djangoCMS**, create a new page at an appropriate location of the page tree. As the page title and slug we should use something describing our product catalog in a way, both meaningful to the customers as well as to search engines.

Next, we change into advanced setting.

As a template we use one with a big placeholder, since it must display our list of products.

As **Application**, select "*Catalog List*" or whatever we named our `ProductsListApp`. This selects the apphook we created in the previous section.

Then we save the page, change into **Structure** mode and locate the Main Content Container. Here we add a container with a Row and Column. As the child of this column we chose a **Catalog List View** plugin from section **Shop**.

Finally we publish the page and enter some text into the search field. Since we haven't assigned any products to the CMS page, we won't see anything yet.

4.5.2 Catalog Detail View

The product's detail pages are the only ones not being managed by the CMS. This is because we often have thousands of products and creating a CMS page for each of them, would be kind of overkill.

Therefore the template used to render the products's detail view is selected automatically by the `ProductRetrieveView` [1] following these rules:

- look for a template named `<myshop>/catalog/<product-model-name>-detail.html` [2] [3], otherwise
- look for a template named `<myshop>/catalog/product-detail.html` [2], otherwise
- use the template `shop/catalog/product-detail.html`.

Use CMS Placeholders on Detail View

If we require CMS functionality for each product's detail page, its quite simple to achieve. To the model class implementing our Product, add djangoCMS Placeholder field named `placeholder`.Then add the templatetag `{% render_placeholder product.placeholder %}` the the template implementing the detail view of our product.

Route requests on Detail View

The `ProductsListApp`, which we previously have registered into **djangoCMS**, is able to route requests on all of its sub-URLs. This is done by expanding the current list of urlpatterns:

Listing 4.3: myshop/urls/products.py

```
from django.conf.urls import patterns, url
from shop.views.catalog import ProductRetrieveView
from myshop.serializers import ProductDetailSerializer

urlpatterns = patterns('',
```

[1] This is the View class responsible for rendering the product's detail view.
[2] `<myshop>` is the app label of the project in lowercase.
[3] `<product-model-name>` is the class name of the product model in lowercase.

```
    # previous patterns
    url(r'^(?P<slug>[\w-]+)$', ProductRetrieveView.as_view(
        serializer_class=ProductDetailSerializer,
    )),
    # other patterns
)
```

All business logic regarding our product now goes into our customized serializer class named `ProductDetailSerializer`. This class then may access the various attributes of our product model and merge them into a serializable representation.

This serialized representation normally requires all attributes from our model, therefore we can write it as simple as:

```
from shop.rest.serializers import ProductDetailSerializerBase

class ProductDetailSerializer(ProductDetailSerializerBase):
    class Meta:
        model = Product
        exclude = ('active',)
```

In case we need *Additional Product Serializer Fields*, lets add them to this class using the serializer fields from the Django RESTFramework library.

Additional Product Serializer Fields

Sometimes such a serializer field shall return a HTML snippet; this for instance is required for image source (`<img src="..." />`) tags, which must thumbnailed by the server when rendered using the appropriate templatetags from the easythumbnail library. For these use cases add a field of type `foo = SerializerMethodField()` with an appropriate method `get_foo()` to our serializer class. This method then may forward the given product to a the built-in renderer:

```
class ProductDetailSerializer(ProductDetailSerializerBase):
    # other attributes

    def get_foo(self, product):
        return self.render_html(product, 'foo')
```

This HTML renderer method looks up for a template following these rules:

- look for a template named `<myshop>/product/catalog-<product-model-name>-<second-argument>.html` [4] [5] [6], otherwise
- look for a template named `<myshop>/product/catalog-product-<second-argument>.html` [4] [6], otherwise
- use the template `shop/product/catalog-product-<second-argument>.html` [6].

Emulate Categories

Since we want to use CMS pages to emulate categories, the product model must declare a relationship between the CMS pages and itself. This usually is done by adding a Many-to-Many field named `cms_pages` to our Product model.

[4] `<myshop>` is the app label of the project in lowercase.
[5] `<product-model-name>` is the class name of the product model in lowercase.
[6] `<field-name>` is the attribute name of the just declared field in lowercase.

Since we work with deferred models, we can not use the mapping table, which normally is generated automatically for Many-to-Many fields by the Django framework. Instead, this mapping table must be created manually and referenced using the though parameter, when declaring the field:

```
from shop.models.product import BaseProductManager, BaseProduct
from shop.models.related import BaseProductPage

class ProductPage(BaseProductPage):
    """Materialize many-to-many relation with CMS pages"""

class Product(BaseProduct):
    # other model fields
    cms_pages = models.ManyToManyField('cms.Page',
        through=ProductPage)

    objects = ProductManager()
```

In this example the class ProductPage is responsible for storing the mapping information between our Product objects and the CMS pages.

Admin Integration

To simplify the declaration of the admin backend used to manage our Product model, **djangoSHOP** is shipped with a special mixin class, which shall be added to the product's admin class:

```
from django.contrib import admin
from shop.admin.product import CMSPageAsCategoryMixin
from myshop.models import Product

@admin.register(Product)
class ProductAdmin(CMSPageAsCategoryMixin, admin.ModelAdmin):
    fields = ('product_name', 'slug', 'product_code',
        'unit_price', 'active', 'description',)
    # other admin declarations
```

This then adds a horizontal filter widget to the product models. Here the merchant must select each CMS page, where the currently edited product shall appear on.

If we are using the method render_html() to render HTML snippets, these are cached by **djangoSHOP**, if caching is configured and enabled for that project. Caching these snippets is highly recommended and gives a noticeable performance boost, specially while rendering catalog list views.

Since we would have to wait until they expire naturally by reaching their expire time, **djangoSHOP** offers a mixin class to be added to the Product admin class, to expire all HTML snippets of a product altogether, whenever a product in saved in the backend. Simply add shop.admin.product.InvalidateProductCacheMixin to the ProductAdmin class described above.

Note: Due to the way keys are handled in many caching systems, the InvalidateProductCacheMixin only makes sense if used in combination with the redis_cache backend.

4.6 Filter Products by its Attributes

Besides *Full Text Search*, adding some filter functionality to an e-commerce site is another very important feature. Customers must be able to narrow down the list of available products to a set of desired products using a combination of prepared filter attributes.

Since in **djangoSHOP** each product class declares its own database model with its own attributes, often related with foreign data models, filtering must be implemented by the merchant on top of the existing product models. Fortunately the REST framework in combination with **‘Django Filter‘_** makes this a rather simple task.

4.6.1 Adding a filter to the List View

In **djangoSHOP** listing the products normally is controlled by `shop.views.catalog.ProductListView` or `shop.views.catalog.CMSPageProductListView`. By default these View classes are configured to use the default filter backends as provided by the REST framework. These filter backends can be configured globally through the settings variable `DEFAULT_FILTER_BACKENDS`.

Additionally we can subclass the filter backends for each View class in our `urls.py`. Say, we need a special catalog filter, which groups our products by a certain product attribute. Then we can create customized filter backend

Listing 4.4: filters.py

```
from rest_framework.filters import BaseFilterBackend

class CatalogFilterBackend(BaseFilterBackend):
    def filter_queryset(self, request, queryset, view):
        queryset = queryset.order_by('attribute__sortmetric')
        return queryset
```

In `urls.py`, where we route requests to the class `shop.views.catalog.ProductListView`, we then replace the default filter backends by our own implementation:

Listing 4.5: myshop/urls/catalog.py

```
from django.conf.urls import patterns, url
from rest_framework.settings import api_settings
from shop.views.catalog import ProductListView
from myshop.serializers import ProductSummarySerializer

urlpatterns = patterns('',
    url(r'^$', ProductListView.as_view(
        serializer_class=ProductSummarySerializer,
        filter_backends=[CatalogFilterBackend],
    ),
)
```

The above example is very simple but gives a rough impression on its possibilities.

Working with Django-Filter

django-filter is a generic, reusable application to alleviate writing some of the more mundane bits of view code. Specifically, it allows users to filter down a queryset based on a model’s fields, displaying the form to let them do this.

REST framework also includes support for generic filtering backends that allow you to easily construct complex searches and filters.

By creating a class which inherit from `django_filters.FilterSet`, we can build filters against each attribute of our product. This filter then uses the passed in query parameters to restrict the set of products available from our catalog:

Listing 4.6: myshop/filters.py

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    width = django_filters.RangeFilter(name='width')
    props = django_filters.MethodFilter(action='filter_properties', widget=SelectMultiple)

    class Meta:
        model = OurProduct
        fields = ['width', 'props']

    def filter_properties(self, queryset, values):
        for value in values:
            queryset = queryset.filter(properties=value)
        return queryset
```

This example assumes that `OurProduct` has a numeric attribute named `width` and a many-to-many field named `properties`.

We then can add this filter to the list view for our products. In **djangoSHOP** we normally do this through the url patterns:

Listing 4.7: myshop/urls.py

```
urlpatterns = patterns('',
    url(r'^$', ProductListView.as_view(
        serializer_class=ProductSummarySerializer,
        filter_class=ProductFilter,
    )),
    # other patterns
)
```

By appending `?props=17` to the URL, the above filter class will restrict the products in our list view to those with a `property` of 17.

4.7 Cascade Plugins

DjangoSHOP extends the eco-system of **djangoCMS** plugins, djangocms-cascade, by additional shop-specific plugins. This allows us to create a whole shopping site, which consists of many different elements, without having to craft templates by hand – with one exception: The product detail views.

Therefore all we have to focus on, is a default page template with one big placeholder. This placeholder then is subdivided into containers, rows, columns and other elements of the Cascade plugin collection.

This however requires a completely different approach, from the designer point of view. The way web design was done a few years ago, starting with the screenshot of a finished page, must be rethought. This has been discussed in length by many web-designers, especially by Brad Frost in his excellent book on Atomic Web Design. He propagates to reverse the design process and start with the smallest entity, which he calls Atoms. They form to bigger components, named Molecules, which themselves aggregate to Organisms.

Some designers nowadays build those components directly in HTML and CSS or SASS, instead of drawing their screens using programs such as InDesign or PhotoShop (which by the way never was intended for this kind of work). It also exempts having the programmer to convert those screens into HTML and CSS – a time consuming and never satisfying job.

According to Frost, the next bigger component after the Organism is the template. This is where **djangocms-cascade** jumps in. Each of the Cascade plugins is shipped with its own default template, which can easily be overwritten by the designers own implementation.

4.7.1 Overriding Templates

For all plugins described here, we can override the provided templates with our own implementation. If the shop framework provides a template, named `/shop/folder/my-organism.html`, then we may override it using `/merchantimplementaion/folder/my-organism.html`.

This template then usually extends the existing framework template with

```
{% extends "/shop/folder/my-organism.html" %}

{% block shop-some-identifier %}
    <div>...</div>
{% endblock %}
```

This is in contrast to Django's own implementation for searching the template, but allows to extend exiting templates more easily.

4.7.2 Breadcrumb

The **BreadcrumbPlugin** has four different rendering options: *Default*, *Soft-Root*, *With Catalog Count* and *Empty*. It can be added exclusively to the placeholder named **Breadcrumb**, unless otherwise configured.

The *Default* breadcrumb behaves as expected. *Soft-Root* appends the page title to the existing breadcrumb, it shall be used for pages marked as soft root. A breadcrumb of type *With Catalog Count* adds a badge containing the number of items. Use an *Empty* to hide the breadcrumb otherwise displayed by the placeholder as default.

4.7.3 Cart

The **CartPlugin** has four different rendering options: Editable, Static, Summary and Watch-List. Refer to the *Cart using a Cascade Plugin* for details.

4.7.4 Checkout Forms

All Forms added to the checkout page are managed by members of the Cascade plugin system. All these plugin inherit from a common base class, `shop.cascade.plugin_base.DialogFormPluginBase`. They all have in common to render and validate one specific Form, which itself inherits from `shop.forms.DialogForm` or `shop.forms.DialogModelForm`.

A nice aspect of this approach is, that ...

- if we add, change or delete attributes in a form, fields are added, changed or deleted from the rendered HTML as well.
- we get client side form validation for free, without having to write any Javascript nor HTML.
- if we add, change or delete attributes in a form, this modification propagates down to both form validation controllers: That one in Javascript used on the client as well as the final one, validating the form on the server.
- if our forms are made out of models, all of the above works as well.
- we can arrange each of those form components using the **Structure** editor from **djangoCMS** toolbar. This is much faster, than by crafting templates manually.

As we can see from this approach, **djangoSHOP** places great value on the principles of a Single Source of Truth, when working with customized database models and forms.

Many of these Forms can be rendered using two different approaches:

Form dialog

Here we render all model fields as input fields and group them into an editable form. This is the normal use case.

Static summary

Here we render all model fields as static strings without wrapping it into a form. This shall be used to summarize all inputs, preferably on the last process step.

These are the currently available plugins provided by **djangoSHOP** to build the checkout page:

Customer Form Plugin

The **Customer Form** is used to query information about some personal information, such as the salutation, the first- and last names, its email address etc. In simple terms, this form combines the fields from the model classes `shop.models.customer.Customer` and `email_auth.models.User` or `auth.models.User` respectively. This means that fields, we add to our `Customer` model, are reflected automatically into this form.

Guest Form Plugin

The **Guest Form** is a reduced version of the **Customer Form**. It only asks for the email address, but nothing else. We use it for customers which do not want to create an account.

Shipping- and Billing Address Forms

There are two form plugins, where customers can add their shipping and/or billing address. The billing address offers a checkbox allowing to reuse the shipping address. By overriding the form templates, this behavior can be switched. Both plugins provide a form made up from the model class implementing `shop.models.address.AddressModel`.

Select the Payment Provider

For each payment provider registered within **djangoSHOP**, this plugin creates a list of radio buttons, where customers can chose their desired payment provider. By overriding the rendering templates, additional forms, for instance to add credit card data, can be added.

Select a Shipping Method

For each shipping provider registered within **djangoSHOP**, this plugin creates a list of radio buttons, where customers can chose their desired shipping method.

Extra Annotations Plugin

This plugin provides a form, where customers can enter an extra annotation, while they proceed through the checkout process.

Accept Condition Plugin

Normally customers must click onto a checkbox to accept various legal requirements, such as the terms and conditions of this site. This plugin offers a text editor, where the merchant can enter a paragraph, possibly with a link onto another CMS page explaining them in more details.

Required Form Fields Plugin

Most checkout forms have one or more required fields. To labels of required input fields, an asterisk is appended. This plugin can be used to add a short text message stating "* These fields are required". It normally should be placed between the last checkout form and the proceed button.

Proceed Button

This plugin adds a styleable proceed button to any placeholder. This kind of button differs from a clickable link button in that sense, that it first sends all gathered form data to the server and awaits a response. Only if all forms are successfully validated, this button proceeds to the given link.

This proceed button can also handle two non-link targets: "Reload Page" and "Purchase Now".

The first target is useful to reload the page in a changed context, for instance if a site visitor logged in and now shall get a personalized page.

The second target is special to **djangoSHOP** and exclusively used, when the customer performs *The Purchasing Operation*.

4.7.5 Authentication

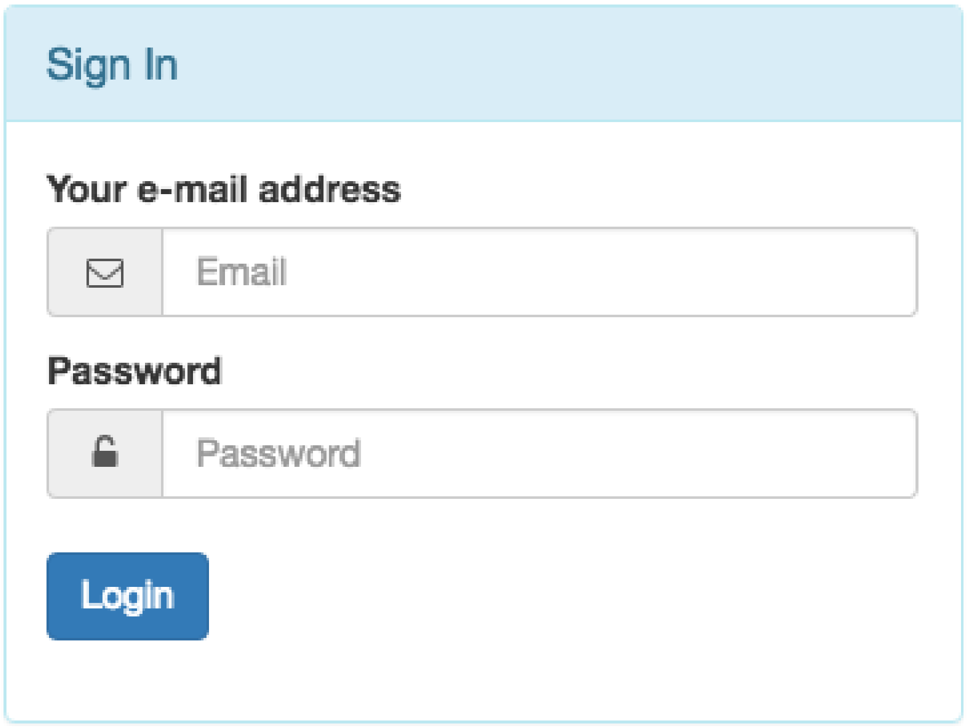
Before proceeding with various input forms, we must know the authentication status of our site visitors. These different states are explained here in detail: *Anonymous Users and Visiting Customers*.

Therefore we need pluggable forms, where visitors can sign in and out, change and rest passwords and so on. All this authentication forms are handled by one single plugin

This plugin handles a bunch of authentication related forms. Lets list them:

Login Form

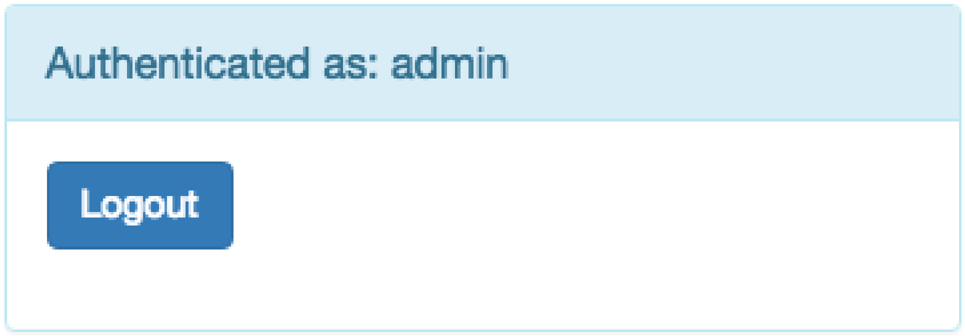
This is a simple login form accepting a username and password.



This form normally is used in combination with **Link type**: *CMS Page*.

Logout Form

This logout form just adds a button to sign out from the site.



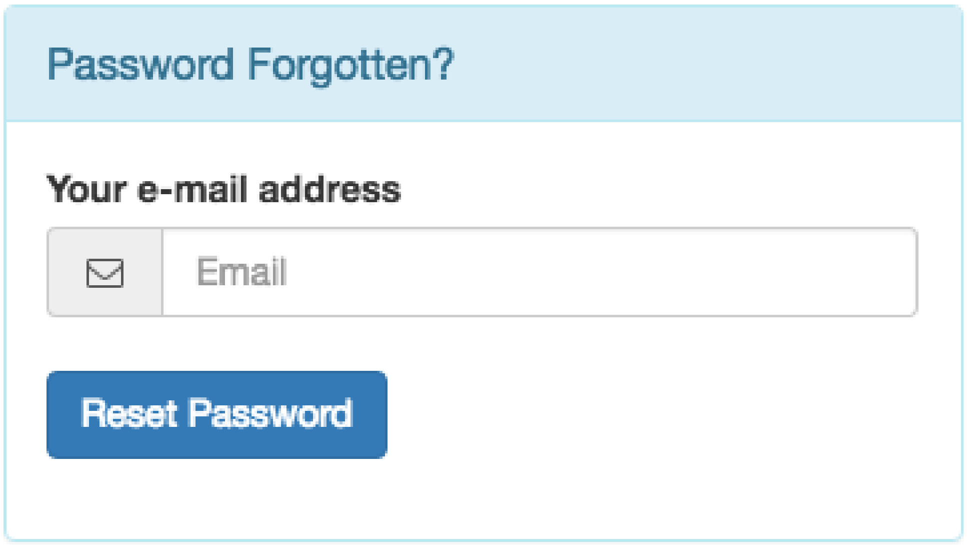
This form normally is used in combination with **Link type**: *CMS Page*.

Shared Login/Logout Form

This combines the *Login Form* with the *Logout Form* so, that anonymous visitors see the login form, while logged in users see the logout form. This form normally is used in combination with **Link type**: *Reload Page*.

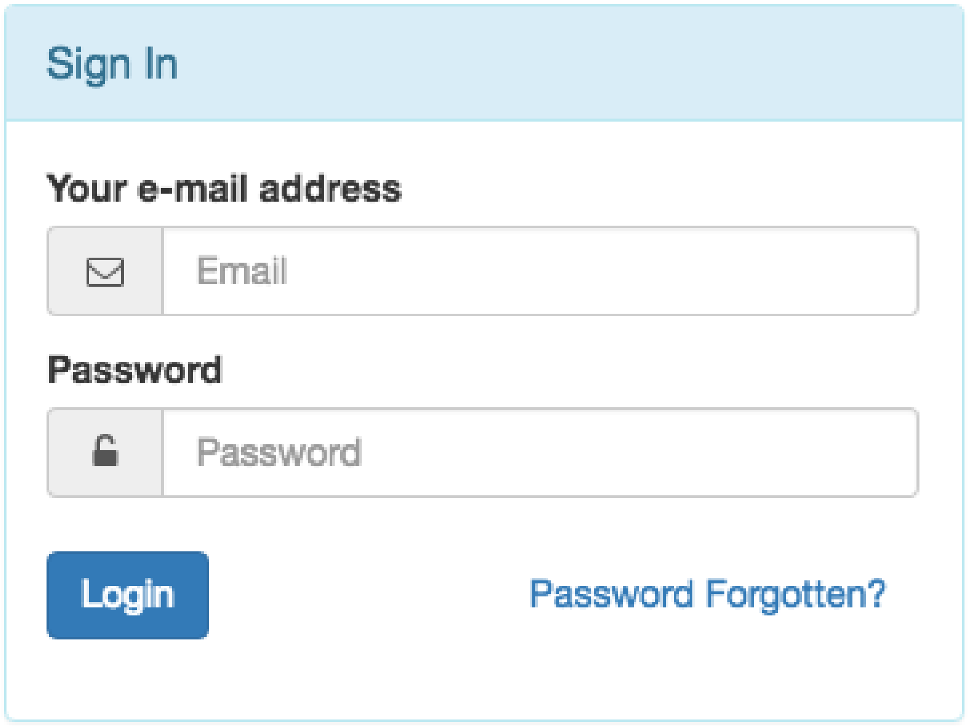
Password Reset Form

This form offers a field, so that registered users, which forgot their password, can enter their email address to start a password reset procedure.

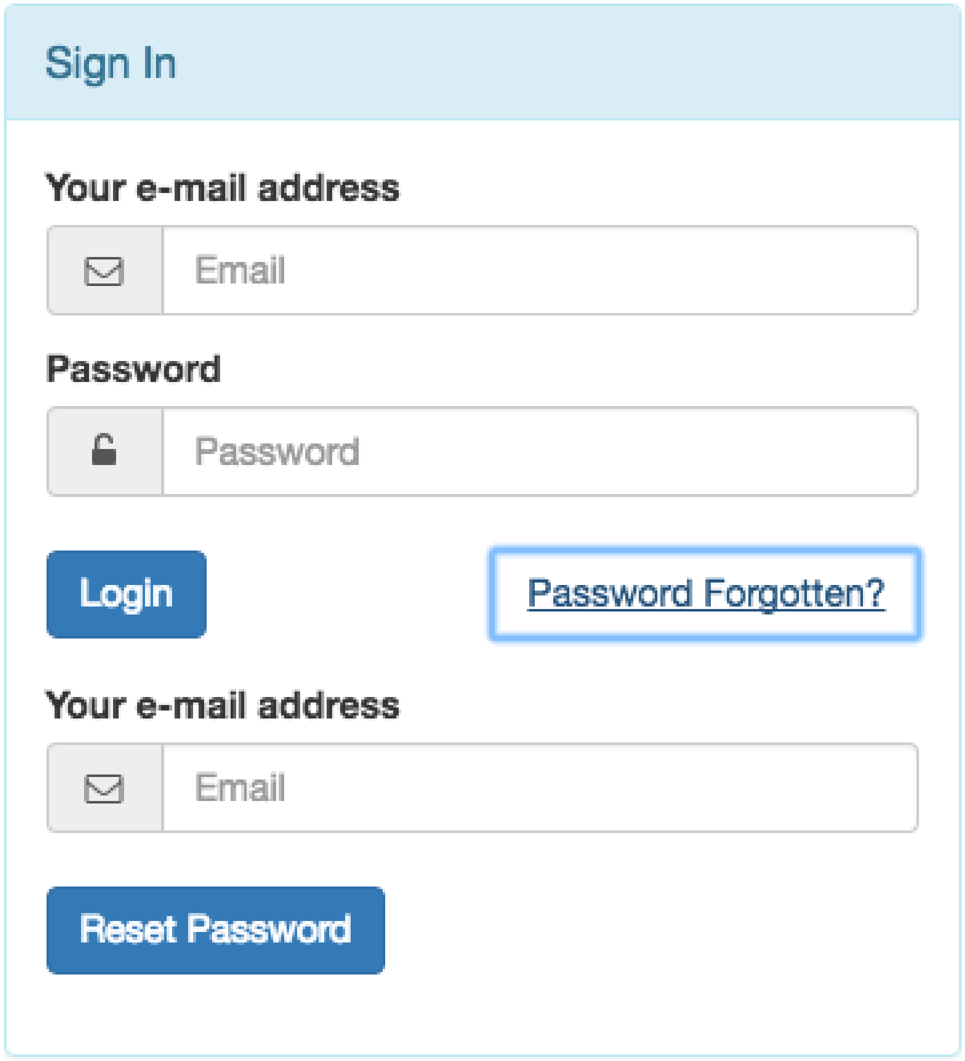


Login & Reset Form

This extends the *Shared Login/Logout Form* by combining it with the *Password Reset Form* form.



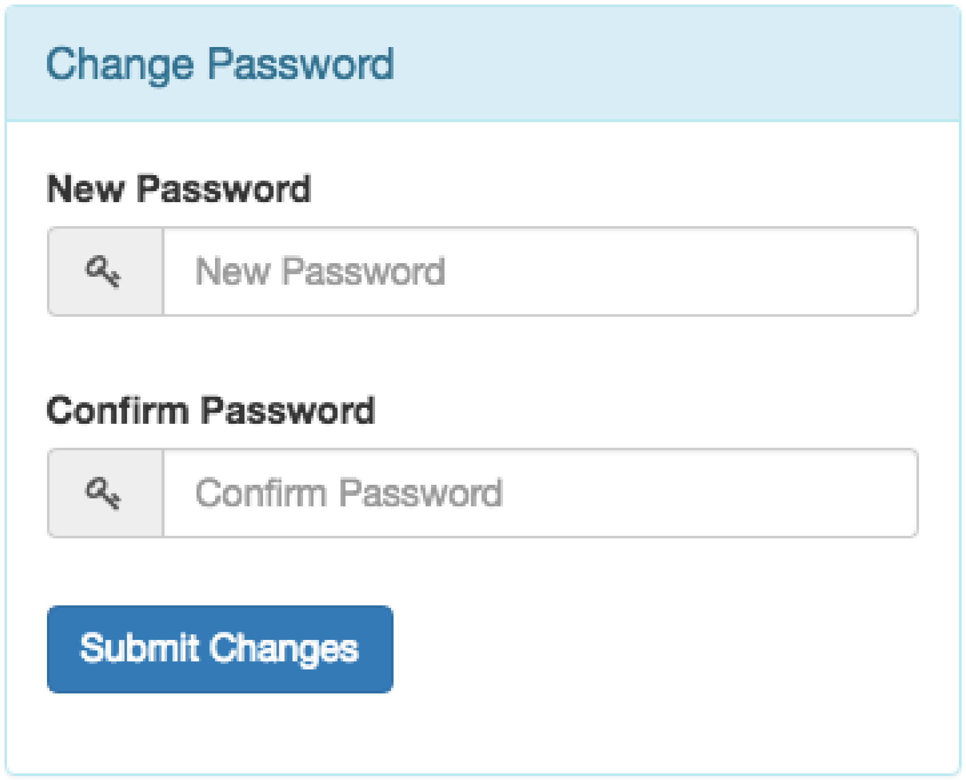
If someone clicks on the link **Password Forgotten?** the form extends to



This form normally is used in combination with **Link type**: *Reload Page*.

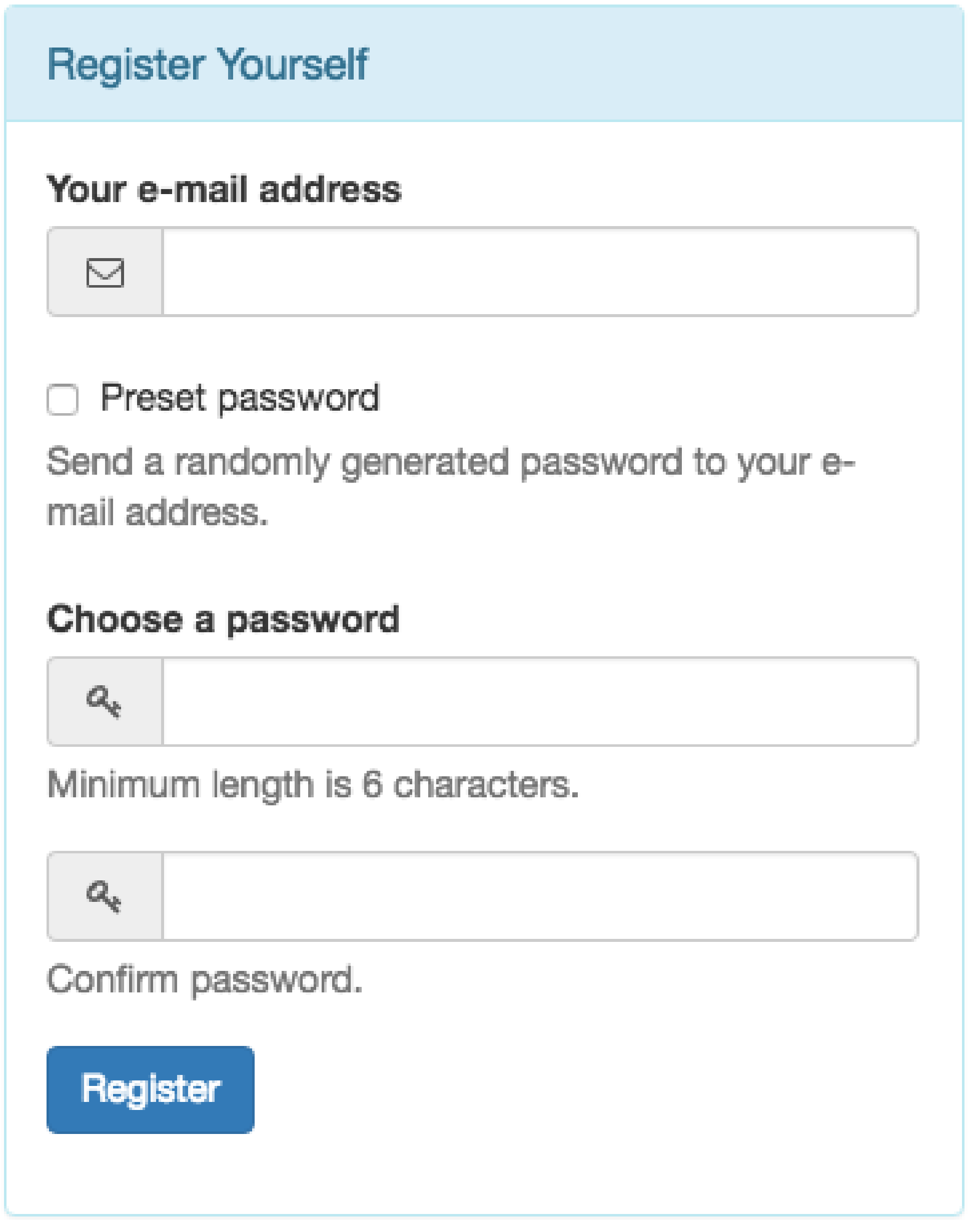
Change Password Form

This form offers two field to change the password. It only appears for logged in users.



Register User Form

Using this form, anonymous visitors can register themselves. After having entered their email address and their desired passwords, they become registered users.


Register Yourself

Your e-mail address

Preset password

Send a randomly generated password to your e-mail address.

Choose a password

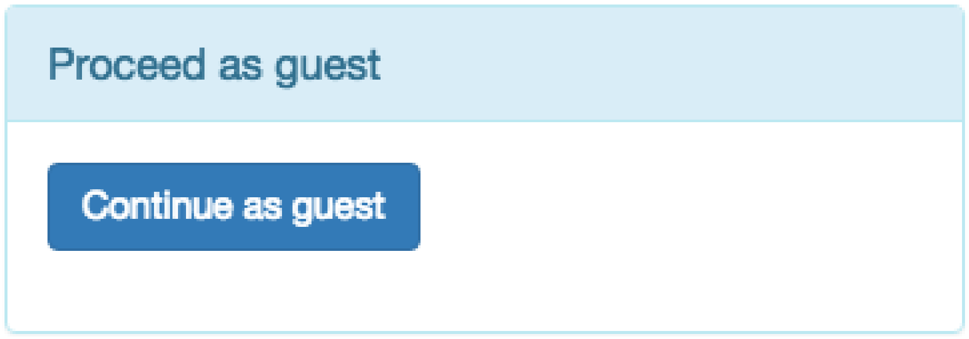
Minimum length is 6 characters.

Confirm password.

This form normally is used in combination with **Link type**: *Reload Page*.

Continue as Guest Form

This form just adds a button, so that visitors can declare themselves as guest users who do not want to register an account, nor expose their identity.

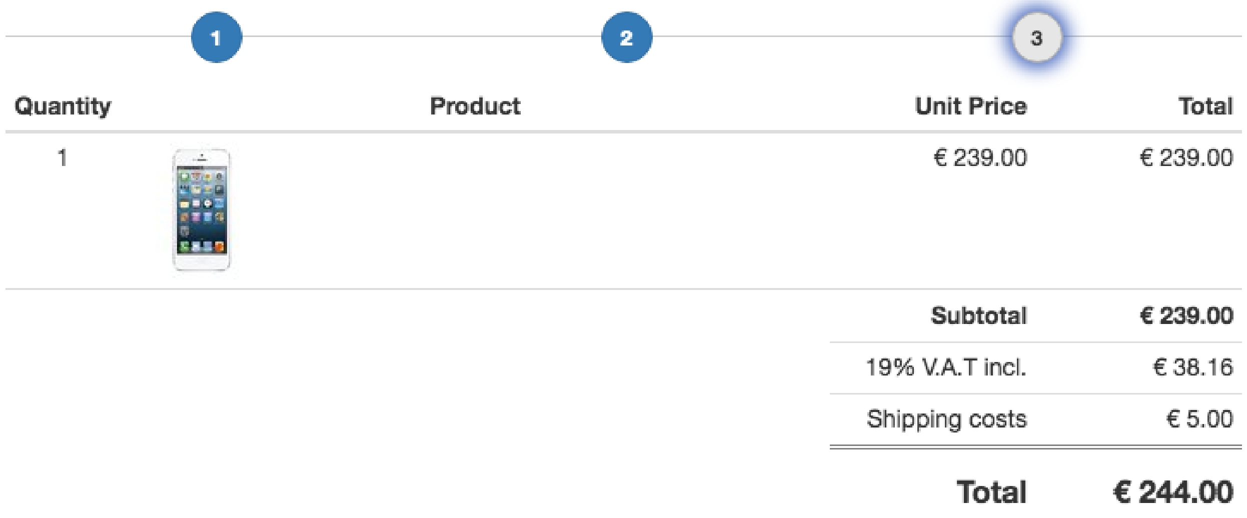


This form normally is used in combination with **Link type**: *Reload Page*.

4.7.6 Process Bar

The **ProcessBarPlugin** can be used to group many forms plugins onto the same page, by dividing them up into different block. Only one block is visible at a time. At to top of that page, a progress bar appears which shows the active step.

This plugin checks the validity of all of its forms and allows to proceed to the next step only, if all of them are valid.



Quantity	Product	Unit Price	Total
1		€ 239.00	€ 239.00
		Subtotal	€ 239.00
		19% V.A.T incl.	€ 38.16
		Shipping costs	€ 5.00
		Total	€ 244.00

Each step in that process bar must contain a **Next Step Button**, so that the customer can move to the next step, provided all forms are valid.

The last step shall contain a *Proceed Button* which shall be configured to take appropriate action, for instance to start the purchasing operation using the **Link type** "*Purchase Now*".

Note: This plugin requires the AngularJS directive `<bsp-process-bar>` as found in the bower package angular-bootstrap-plus.

4.7.7 Catalog

The catalog list view is handled by the **ShopCatalogPlugin**.

This plugin requires a CMS page, which uses the apphook **ProductsListApp**. First assure that we *Create the ProductsListApp*. This CMSapp must be implemented by the merchant; it thus is part of the project, rather than the **djangoSHOP** framework.

4.7.8 Viewing Orders

The **Order Views** plugin is used to render the list- and detail views of orders, specific to the currently logged in customer. Without a number in the URL, a list of all orders belonging to the current customer is shown. By adding the primary key of a specific order to the URL, all ordered items from that specific order are shown. We name this the order detail view, although it is a list of items.

This plugin requires a CMS page, which as uses the CMSApp **OrderApp**. This CMS application is part of the shop framework and always available in the *Advanced Settings* of each CMS page.

The Order List- and Detail Pages share one common entity in our CMS page tree. The Order Detail view just rendered in a different way. Editing this pseudo page therefore is not possible because it is not part of the CMS.

4.7.9 Search Results

Rendering search results is handled by the **Search Results** plugin.

On a site offering full-text search, add a page to display search results. First assure that we have a *Search View* assigned to that page as apphook. This CMSapp must be implemented by the merchant; it thus is part of the project, rather than the **djangoSHOP** framework.

4.8 Cart and Checkout

In **djangoSHOP** the cart's content is always stored inside the database. In previous versions of the software, the cart's content was kept inside the session for anonymous users and stored in the database for logged in users. Now the cart is always stored in the database. This approach simplifies the code and saves some random access memory, but adds another minor problem:

From a technical point of view, the checkout page is the same as the cart. They can both be on separate pages, or be merged on the same page. Since what we would normally name the "*Checkout Page*", is only a collection of *Cascade Plugins*, we won't go into further detail here.

4.8.1 Expired Carts

Sessions expire, but then the cart's content of anonymous customers still remains in the database. We therefore must assure that these carts will expire too, since they are of no use for anybody, except maybe for some data-mining.

By invoking

```
./manage.py shopcustomers
Customers in this shop: total=3408, anonymous=140, expired=88,
    active=1108, guests=2159, registered=1109, staff=5.
```

we gather some statistics about former visiting customers of our **djangoSHOP**. Here we see that 1109 customers bought as registered users, while 2159 bought as guests. There are 88 customers in the database, but they don't have any associated session anymore, hence they can be considered as expired. Invoking

```
./manage.py shopcustomers --delete-expired
```

deletes those expired customers, and with them their expired carts. This task shall be performed by a cronjob on a daily basis.

4.8.2 Cart Models

The cart consists of two models classes `Cart` and `CartItem`, both inheriting from `BaseCart` and `BaseCartItem` respectively. As with most models in **djangoSHOP**, these are using the *Deferred Model Pattern*, so that inheriting from a base class automatically sets the foreign keys to the appropriate model. This gives the programmer the flexibility to add as many fields to the cart, as the merchant requires for his special implementation.

In most use-cases, the default cart implementation will do the job. These default classes can be found at `shop.models.defaults.cart.Cart` and `shop.models.defaults.cart_item.CartItem`. To materialize the default implementation, it is enough to `import` these two files into the merchants shop project. Otherwise we create our own cart implementation inheriting from `BaseCart` and `BaseCartItem`. Since the item quantity can not always be represented by natural numbers, this field must be added to the `CartItem` implementation rather than its base class. Its field type must be countable, so only `IntegerField`, `FloatField` or `DecimalField` are allowed as quantity.

Note: Assure that the model `CartItem` is imported (and materialized) before model `Product` and classes derived from it.

The `Cart` model uses its own manager. Since there is only one cart per customer, accessing the cart must be performed using the `request` object. We can always access the cart for the current customer by invoking:

```
from shop.models.cart import CartManager

cart = CartManager.get_or_create_from_request(request)
```

Adding a product to the cart, must be performed by invoking:

```
from shop.models.cart import CartItemManager

cart_item = CartItemManager.get_or_create(cart=cart,
        product=product, quantity=quantity, **extras)
```

This returns a new cart item object, if the given product could not be found in the current cart. Otherwise it returns the existing cart item, increasing the quantity by the given value. For products with variations it’s not always trivial to determine if they shall be considered as existing cart items, or as new ones. Since **djangoSHOP** can’t tell that difference for any kind of product, it delegates this question. Therefore the class implementing the shop’s products shall override their method `is_in_cart`. This method is used to tell the `CartItemManager` whether a product has already been added to the cart or is new.

Whenever the method `cart.update(request)` is invoked, the cart modifiers run against all items in the cart. This updates the line totals, the subtotal, extra costs and the final sum.

Watch List

Instead of implementing a separate watch-list (some would say wish-list), **djangoSHOP** uses a simple trick. Whenever the quantity of a cart item is zero, this item is considered to be in the watch list. Otherwise it is considered to be in the cart. The train of though is as follows: A quantity of zero, never makes sense for items in the cart. On the other side, any quantity makes sense for items in the watch-list. Therefore reducing the quantity of a cart item to zero is the same as keeping an eye on it, without actually wanting it to purchase.

4.8.3 Cart Views

Displaying the cart in **djangoSHOP** is as simple, as adding any other page to the CMS. Change into the Django admin backend and enter into the CMS page tree. At an appropriate location in that tree add a new page. As page title use "Cart", "Basket", "Warenkorb", "Cesta", or whatever is appropriate in the natural language used for that site. Multilingual CMS installations offer a page title for each language.

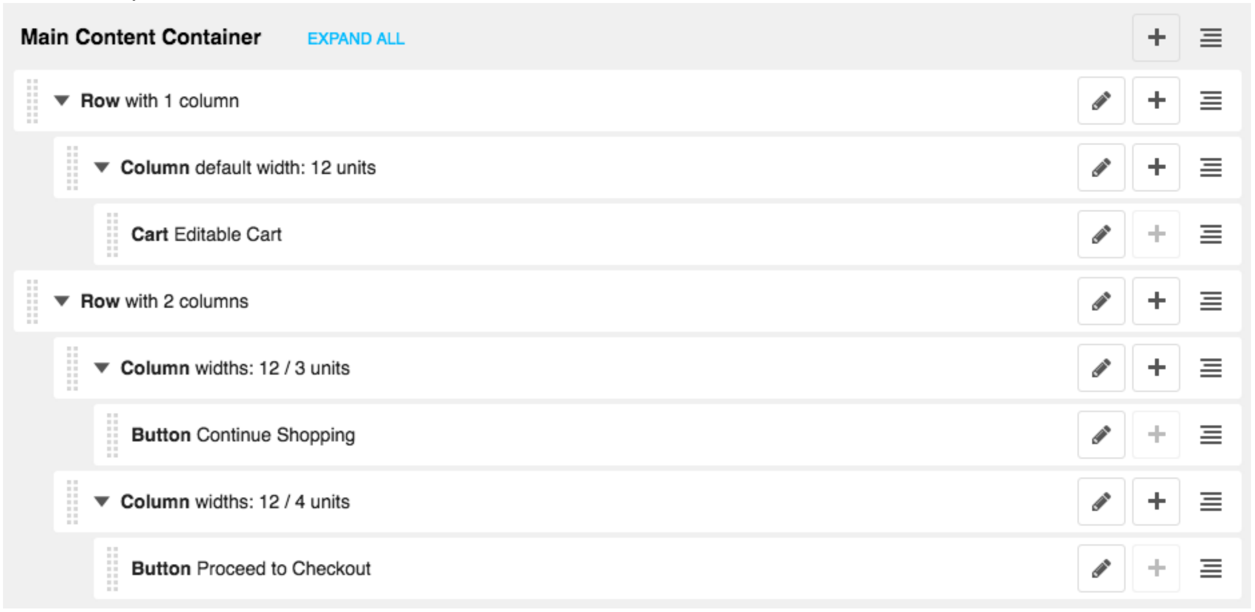
In the CMS page editor click onto the link named **Advanced Settings** at the bottom of the popup window. As template, chose the default one, provided it contains at least one big placeholder.

Enter "shop-cart" into the **Id**-field just below. This identifier is required by some templates which link directly onto the cart view page. If this field is not set, some links onto the cart page might not work properly.

It is suggested to check the checkbox named **Soft root**. This prevents that a menu item named "Cart" will appear side by side with other pages from the CMS. Instead, we prefer to render a special cart symbol located on the right of the navigation bar.

Cart using a Cascade Plugin

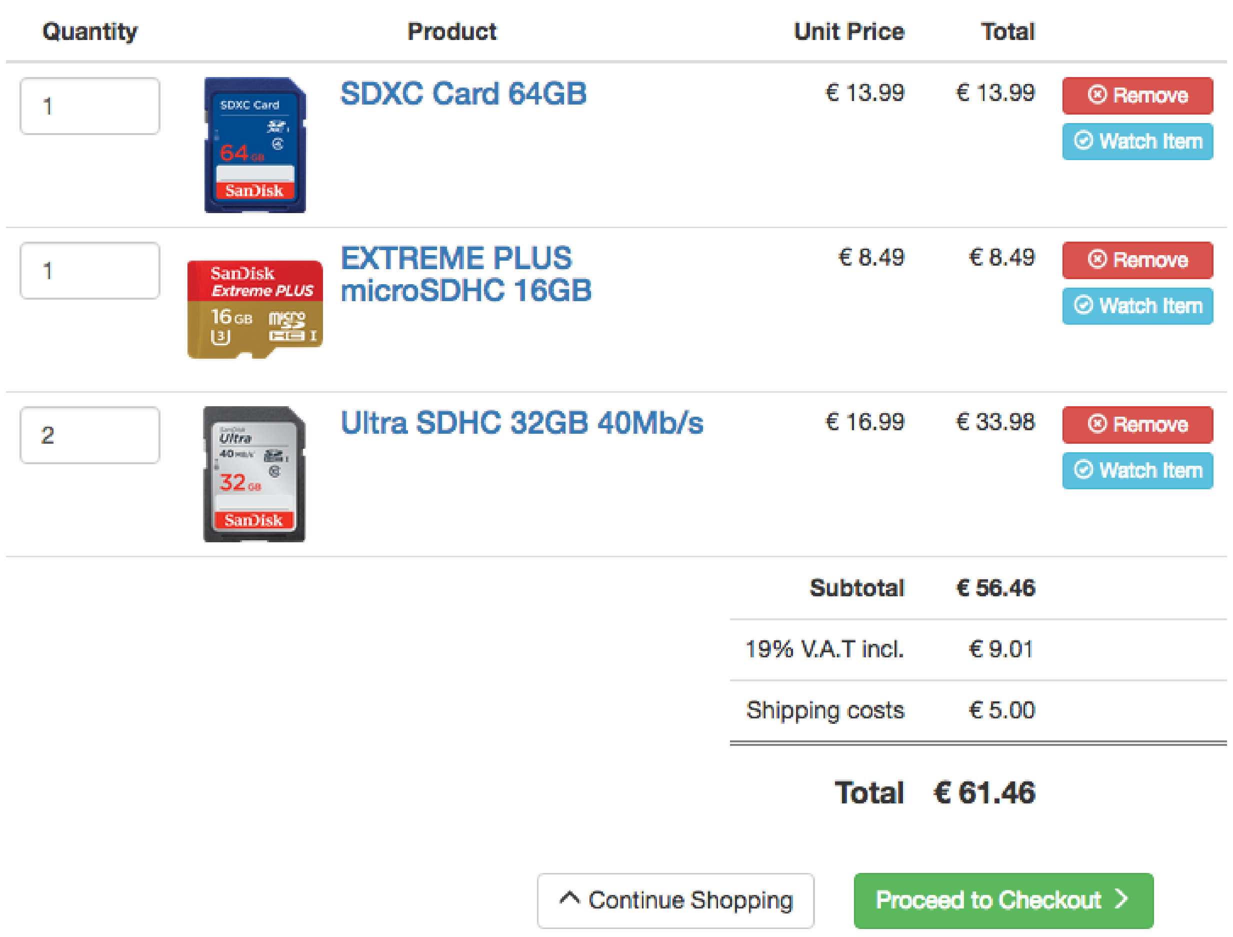
Click onto **View on site** and change into front-end editing mode to use the grid-system of djangocms-cascade. Locate the main placeholder and add a **Row** followed by at least one **Column** plugin; both can be found in section **Bootstrap**. Below that column plugin, add a child named **Cart** from section **Shop**. This Cart Plugin can be rendered in four different ways:


Main Content Container
EXPAND ALL
Row with 1 column
Column default width: 12 units
Cart Editable Cart
Row with 2 columns
Column widths: 12 / 3 units
Button Continue Shopping
Column widths: 12 / 4 units
Button Proceed to Checkout

Editable Cart

An "Editable Cart" is rendered using the Angular JS template engine. This means that a customer may change the number of items, delete them or move them the the watch-list. Each update is reflected immediately into the cart's subtotal, extra fields and final totals.

Using the above structure, the rendered cart will look similar to this.



Quantity		Product	Unit Price	Total	
1		SDXC Card 64GB	€ 13.99	€ 13.99	Remove / Watch Item
1		EXTREME PLUS microSDHC 16GB	€ 8.49	€ 8.49	Remove / Watch Item
2		Ultra SDHC 32GB 40Mb/s	€ 16.99	€ 33.98	Remove / Watch Item
			Subtotal	€ 56.46	
			19% V.A.T incl.	€ 9.01	
			Shipping costs	€ 5.00	
			Total	€ 61.46	

Depending on the chosen template, this layout may vary.

Static Cart

An alternative to the editable cart is the 'static cart'. Here the cart items are rendered by the Django template engine. Since here everything is static, the quantity can't be changed anymore and the customer would have to proceed to the checkout without being able to change his mind. This probably only makes sense when purchasing a single product.

Cart Summary

This only displays the cart's subtotal, the extra cart fields, such as V.A.T., shipping costs and the final total.

Watch List

A special view of the cart is the watch list. It can be used by customers to remember items they want to compare or buy sometimes later. The watch-list by default is editable, but does not allow to change the quantity. This is because the watch-list shares the same object model as the cart items. If the quantity of an item 0, then that cart item is considered to be watched. If instead the quantity is 1 ore more, the item is considered to be in the cart. It therefore is very easy to move items from the cart to the watch-list and vice versa. This concept also disallows to have an item in both the cart and the watch-list. This during online shopping, often can be a major point of confusion.

Render templates

The path of the templates used to render the cart views is constructed using the following rules:

- Look for a folder named according to the project's name, ie. `settings.SHOP_APP_LABEL` in lower case. If no such folder can be found, then use the folder named `shop`.
- Search for a subfolder named `cart`.
- Search for a template named `editable.html`, `static.html`, `watch.html` or `summary.html`.

These templates are written to be easily extensible by the customized templates. To override the "*editable cart*" add a template with the path, say `myshop/cart/editable.html` to the projects template folder. This template then shall begin with `{% extend "shop/cart/editable.html" %}` and only override the `{% block %}...{% endblock %}` interested in.

Many of these template blocks are themselves embedded inside HTML elements such as `<script id="shop/....html" type="text/ng-template">`. The reason for this is that the editable cart is rendered in the browser by AngularJS using so called directives. Hence it becomes very straight-forward to override Angular's script templates using Django's internal template engine.

Multiple templates If for some special reasons we need different cart templates, then we must add this line to the projects `settings.py`:

```
CMSPLUGIN_CASCADE_PLUGINS_WITH_EXTRA_RENDER_TEMPLATES = {
    'ShopCartPlugin': (
        (None, _("default")),  # the default behavior
        ('myproject/cart/other-editable.html', _("extra editable")),
    )
}
```

This will add an extra select button to the cart editor. The site administrator then can chose between the default template and an extra editable cart template.

Proceed to Checkout On the cart's view, the merchant may decide whether to implement the checkout forms together with the cart, or to create a special checkout page onto which the customer can proceed. From a technical point of view, it doesn't make any difference, if the cart and the checkout are combined on the same CMS page, or if they are split across two or more pages. In the latter case simply add a button at the end of each page, so that the customer can easily proceed to the next one.

On the checkout page, the customer has to fill out a few forms. These can be a contact form, shipping and billing addresses, payment and shipping methods, and many more. Which ones depend on the configuration, the legal regulations and the requirements of the shop's implementation. In *Cascade Plugins* all shop specific CMS plugins are listed. They can be combined into whatever makes sense for a successful checkout.

Add a Cart via manually written Cart Template

Instead of using the CMS plugin system, the template for the cart can also be implemented manually. Based on an existing page template, locate the element, where the cart shall be inserted. Then use one of the existing templates in the folder `django-shop/shop/templates/shop/cart/` as a starting point, and insert it at an appropriate location in the page template. Next, in the project's `settings.py`, add this specialized template to the list `CMS_TEMPLATES` and select it for that page.

From a technical point of view, it does not make any difference whether we use the cart plugin or a handcrafted template. If the HTML code making up the cart has to be adopted to the merchants needs, we normally are better off and much more flexible, if we override the template code as described in section *Render templates*. Therefore, it is strongly discouraged to craft cart and checkout templates by hand.

4.8.4 Cart Modifiers

Cart Modifiers are simple plugins that allow the merchant to define rules in a programmatic way, how the totals of a cart are computed and how they are labeled. A typical job is to compute tax rates, adding discounts, shipping and payment costs, etc.

Instead of implementing each possible combination for all of these use cases, the **djangoSHOP** framework offers an API, where third party applications can hooks into every computational step. One thing to note here is that Cart Modifiers are not only invoked, when the cart is complete and the customer wants to proceed to the checkout, but also for each item before being added to the cart.

This allows the programmer to vary the price of certain items, depending on the current state of the cart. It can for instance be used, to set one price for the first item, and other prices for every further items added to the cart.

Cart Modifiers are split up into three different categories: Generic, Payment and Shipping. In the shops `settings.py` they must be configured as a list or tuple such as:

```
SHOP_CART_MODIFIERS = (
    'shop.modifiers.defaults.DefaultCartModifier',
    'shop.modifiers.taxes.CartExcludedTaxModifier',
    'myshop.modifiers.PostalShippingModifier',
    'shop.modifiers.defaults.PayInAdvanceModifier',
    'shop_stripe.modifiers.StripePaymentModifier',
)
```

When updating the cart, these modifiers are applied in the order of the above list. Therefore it makes a difference, if taxes are applied before or after having applied the shipping costs.

Moreover, whenever in the detail view the quantity of a product is updated, then all configured modifiers are ran for that item. This allows the `ItemModelSerializer`, to even change the unit price of product depending on the total content of the cart.

Cart modifiers are easy to write and they normally consist only of a few lines of code. It is the intention of **djangoSHOP** to seed an eco-system for these kinds of plugins.

Here is an incomplete list of some useful cart modifiers.

Generic Cart Modifiers

These kinds of cart modifiers are applied unconditionally onto the cart. A typical instance is the `DefaultCartModifier`, the `CartIncludeTaxModifier` or the `CartExcludeTaxModifier`.

DefaultCartModifier

The `shop.modifiers.default.DefaultCartModifier` is required for almost every shopping cart. It handles the most basic calculations, ie. multiplying the items unit prices with the chosen quantity. Since this modifier sets the cart items line total, it must be listed as the first entry in `SHOP_CART_MODIFIERS`.

Payment Cart Modifier

From these kinds of modifiers, only that for the chosen payment method is applied. Payment Modifiers are used to add extra costs or discounts depending on the chosen payment method. By overriding the method `is_disabled` a payment method can be disabled; useful to disable certain payments in case the carts total is below a certain threshold.

Shipping Cart Modifier

From these kinds of modifiers, only that for the chosen shipping method is applied. Shipping Modifiers are used to add extra costs or discounts depending on chosen shipping method, the number of items in the cart and their weight. By overriding the method `is_disabled` a shipping method can be disabled; useful to disable certain payments in case the carts total is below a certain threshold.

How Modifiers work

Cart modifiers should extend the *`shop.modifiers.base.BaseCartModifier`* class and extend one or more of the given methods:

Note: Until version 0.2 of **djangoSHOP**, the Cart Modifiers returned the amount and label for the extra item rows, and **djangoSHOP** added them up. Since Version 0.3 cart modifiers must change the line subtotals and cart total themselves.

class `shop.modifiers.base.`**`BaseCartModifier`**(*identifier=None*)

Cart Modifiers are the cart's counterpart to backends.

It allows to implement taxes and rebates / bulk prices in an elegant and reusable manner: Every time the cart is refreshed (via it's update() method), the cart will call all subclasses of this modifier class registered with their full path in *settings.SHOP_CART_MODIFIERS*.

The methods defined here are called in the following sequence: 1. *pre_process_cart*: Totals are not computed, the cart is "rough": only relations and quantities are available 1a. *pre_process_cart_item*: Line totals are not computed, the cart and its items are "rough": only relations and quantities are available 2. *process_cart_item*: Called for each cart_item in the cart. The modifier may change the amount in *cart_item.line_total*. 2a. *add_extra_cart_item_row*: It optionally adds an object of type *ExtraCartRow* to the current cart item. This object adds additional information displayed on each cart items line. 3. *process_cart*: Called once for the whole cart. Here, all fields relative to cart items are filled. Here the carts subtotal is used to computer the carts total. 3a. *add_extra_cart_row*: It optionally adds an object of type *ExtraCartRow* to the current cart. This object adds additional information displayed in the carts footer section. 4. *post_process_cart*: all totals are up-to-date, the cart is ready to be displayed. Any change you make here must be consistent!

Each method accepts the HTTP *request* object. It shall be used to let implementations determine their prices according to the session, and other request information. The *request* object also can be used to store arbitrary data to be passed between modifers using the temporary dict *request.cart_modifiers_state*.

`add_extra_cart_item_row`(*cart_item*, *request*)

Optionally add an *ExtraCartRow* object to the current cart item.

This allows to add an additional row description to a cart item line. This method optionally utilizes or modifies the amount in *cart_item.line_total*.

`add_extra_cart_row`(*cart*, *request*)

Optionally add an *ExtraCartRow* object to the current cart.

This allows to add an additional row description to the cart. This method optionally utilizes *cart.subtotal* and modifies the amount in *cart.total*.

`arrange_cart_items`(*cart_items*, *request*)

Arrange all items, which are intended for the shopping cart. Override this method to resort and regroup the returned items.

arrange_watch_items (*watch_items*, *request*)
Arrange all items, which are being watched. Override this method to resort and regroup the returned items.

post_process_cart (*cart*, *request*)
This method will be called after the cart was processed in reverse order of the registered cart modifiers. The Cart object is "final" and all the fields are computed. Remember that anything changed at this point should be consistent: If updating the price you should also update all relevant totals (for example).

pre_process_cart (*cart*, *request*)
This method will be called before the Cart starts being processed. It shall be used to populate the cart with initial values, but not to compute the cart's totals.

pre_process_cart_item (*cart*, *item*, *request*)
This method will be called for each item before the Cart starts being processed. It shall be used to populate the cart item with initial values, but not to compute the item's linetotal.

process_cart (*cart*, *request*)
This will be called once per Cart, after every line item was treated by method *process_cart_item*.

The subtotal for the cart is already known, but the total is still unknown. Like for the line items, the total is expected to be calculated by the first cart modifier, which typically is the *DefaultCartModifier*. Posterior modifiers can optionally change the total and add additional information to the cart using an object of type *ExtraCartRow*.

process_cart_item (*cart_item*, *request*)
This will be called for every line item in the Cart: Line items typically contain: product, unit_price, quantity and a dictionary with extra row information.

If configured, the starting line total for every line (unit price * quantity) is computed by the *DefaultCart-Modifier*, which typically is listed as the first modifier. Posterior modifiers can optionally change the cart items line total.

After processing all cart items with all modifiers, these line totals are summed up to form the carts subtotal, which is used by method *process_cart*.

4.9 Payment Providers

Payment Providers are simple classes, which create an interface from an external Payment Service Provider (shortcut PSP) to our **djangoSHOP** framework.

Payment Providers must be aggregates of a *Payment Cart Modifier*. Here the Payment Cart Modifier computes extra fees when selected as a payment method, whereas our Payment Provider class, handles the communication with the configured PSP, whenever the customer submits the purchase request.

In **djangoSHOP** Payment Providers normally are packed into separate plugins, so here we will show how to create one yourself instead of explaining the configuration of an existing Payment gateway.

A precautionary measure during payments with credit cards is, that the used e-commerce implementation never sees the card numbers or any other sensible information. Otherwise those merchants would have to be PCI-DSS certified, which is an additional, but often unnecessary bureaucratic task, since most PSPs handle that task for us.

4.9.1 Checkout Forms

Since the merchant is not allowed to "see" sensitive credit card information, some Payment Service Providers require, that customers are redirected to their site so that there, they can enter their credit card numbers. This for some customers is disturbing, because they visually leave the current shop site.

Therefore other PSPs allow to create form elements in HTML, whose content is send to their site during the purchase task. This can be done using a POST submission, followed by a redirection back to the client. Other providers use Javascript for submission and return a payment token to the customer, who himself forwards that token to the shopping site.

All in all, there are so many different ways to pay, that it is quite tricky to find a generic solution compatible for all of them.

Here **djangoSHOP** uses some Javascript during the purchase operation. Lets explain how:

The Purchasing Operation

During checkout, the clients final step is to click onto a button labeled something like “Buy Now”. This button belongs to an AngularJS controller, provided by the directive `shop-dialog-proceed`. It may look similar to this:

```
<button shop-dialog-proceed ng-click="proceedWith('PURCHASE_NOW')" class="btn btn-success">Buy Now</b
```

Whenever the customer clicks onto that button, the function `proceedWith('PURCHASE_NOW')` is invoked in the scope of the AngularJS controller, belonging to the given directive.

This function first uploads the current checkout forms to the server. There they are validated, and if everything is OK, an updated checkout context is send back to the client. See `shop.views.checkout.CheckoutViewSet.upload()` for details.

Next, the success handler of the previous submission looks at the given action. In `proceedWith`, we used the magic keyword `PURCHASE_NOW`, which starts a second submission to the server, requesting to begin with the purchase operation (See `shop.views.checkout.CheckoutViewSet.purchase()` for details.). This method determines he payment provider previously chosen by the customer. It then invokes the method `get_payment_request()` of that provider, which returns a Javascript expression.

On the client, this returned Javascript expression is passed to the eval() function and executed; it then normally starts to submit the payment request, sending all credit card data to the given PSP.

While processing the payment, PSPs usually need to communicate with the shop framework, in order to inform us about success or failure of the payment. To communicate with us, they may need a few endpoints. Each Payment provider may override the method `get_urls()` returning an `urlpattern`, which then is used by the Django URL resolving engine.

```
class MyPSP(PaymentProvider):
    namespace = 'my-psp-payment'

    def get_urls(self):
        urlpatterns = patterns('',
            url(r'^success$', self.success_view, name='success'),
            url(r'^failure$', self.failure_view, name='failure'),
        )
        return urlpatterns

    def get_payment_request(self, cart, request):
        js_expression = 'scope.charge().then(function(response) { $window.location.href=response.data
        return js_expression

    @classmethod
    def success_view(cls, request):
        # approve payment using request data returned by PSP
        cart = CartModel.objects.get_from_request(request)
        order = OrderModel.objects.create_from_cart(cart, request)
        order.add_paypal_payment(payment.to_dict())
        order.save()
        thank_you_url = OrderModel.objects.get_latest_url()
        return HttpResponseRedirect(thank_you_url)

    @classmethod
    def failure_view(cls, request):
        """Redirect onto an URL informing the customer about a failed payment"""
        cancel_url = Page.objects.public().get(reverse_id='cancel-payment').get_absolute_url()
        return HttpResponseRedirect(cancel_url)
```

Note: The directive `shop-dialog-proceed` evaluates the returned Javascript expression inside a chained `then(...)`-handler from the AngularJS promise framework. This means that such a function may itself return a new promise, which is resolved by the next `then()`-handler.

As we can see in this example, by evaluating arbitrary Javascript on the client, combined with HTTP-handlers for any endpoint, **djangoSHOP** is able to offer an API where adding new Payment Service Providers doesn't require any special tricks.

4.10 Order

During checkout, at a certain point the customer has to click on a button named "*Purchase Now*". This operation performs quite a few tasks, one of them is to convert the cart with its items into an order. The final task is to reset the cart, which means to remove its content. This operation is atomic and not reversible.

4.10.1 Order Models

An order consists of two models classes `Order` and `OrderItem`, both inheriting from `BaseOrder` and `BaseOrderItem` respectively. As with most models in **djangoSHOP**, they are *Deferred Model Pattern*, so that inheriting from a base class automatically sets the foreign keys to the appropriate model. This gives the programmer the flexibility to add as many fields to the order, as the merchant requires for his special implementation.

In most use-cases, the default order implementation will do the job. These default classes can be found at `shop.models.defaults.order.Order` and `shop.models.defaults.order_item.OrderItem`. To materialize the default implementation, it is enough to `import` these two files into the merchants shop project. Otherwise the programmer may create his own order implementation inheriting from `BaseOrder` and/or `BaseOrderItem`.

Note: Assure that the model `OrderItem` is imported (and materialized) before model `Product` and classes derived from it.

The order item quantity can not always be represented by natural numbers, therefore this field must be added to the `OrderItem` implementation rather than its base class. Since the quantity is copied from the cart item to the order item, its field type must must correspond to that of `CartItem.quantity`.

Create an Order from the Cart

Whenever the customer performs the purchase operation, the cart object is converted into a new order object by invoking:

```
from shop.models.order import OrderModel

order = OrderModel.objects.create_from_cart(cart, request)
```

This operation is atomic and can take some time. It normally is performed by the payment provider, whenever a successful payment was received.

Since the merchants implementation of `Cart`, `CartItem`, `Order` and `OrderItem` may contain extra fields the shop framework isn't aware of, these fields have to be converted from the cart to the order objects during the purchasing operation.

If required the merchant's implementation of `Order` shall override the method `populate_from_cart(cart, request)`, which provides a hook to copy those extra fields from the cart object to the order object.

Similarly the merchant's implementation of `OrderItem` shall override the method `populate_from_cart_item(cart_item, request)`, which provides a hook to copy those extra fields from the cart item to the order item object.

Order Numbers

In commerce it is mandatory that orders are numbered using a unique and continuously increasing sequence. Each merchant has his own way to generate this sequence numbers and in some implementations it may even come from an external generator, such as an ERP system. Therefore **djangoSHOP** does not impose any numbering scheme for the orders. This intentionally is left over to the merchant's implementation.

Each Order model must implement two methods, one to create and and one to retrieve the order numbers. A simple implementation may look like this:

```
from django.db import models
from django.utils.datetime_safe import datetime
from shop.models import order

class Order(order.BaseOrder):
    number = models.PositiveIntegerField("Order Number", null=True, default=None, unique=True)

    def get_or_assign_number(self):
        if self.number is None:
            epoch = datetime.now().date()
            epoch = epoch.replace(epoch.year, 1, 1)
            qs = Order.objects.filter(number__isnull=False, created_at__gt=epoch)
            qs = qs.aggregate(models.Max('number'))
            try:
                epoc_number = int(str(qs['number__max'])[4:]) + 1
                self.number = int('{0}{1:05d}'.format(epoch.year, epoc_number))
            except (KeyError, ValueError):
                # the first order this year
                self.number = int('{0}00001'.format(epoch.year))
        return self.get_number()

    def get_number(self):
        return '{0}-{1}'.format(str(self.number)[:4], str(self.number)[4:])
```

Here the first four digits specify the year in which the order was generated, whereas the last five digits are a continuous increasing sequence.

4.10.2 Order Views

Displaying the last or former orders in **djangoSHOP** is as simple, as adding two pages to the CMS. Change into the Django admin backend and enter into the CMS page tree. At an appropriate location in that tree add a new page. As page title use “My Orders”, “Ihre Bestellungen”, “Mis Pedidos”, or whatever is appropriate in the natural language used for that site. Multilingual CMS installations offer a page title for each language.

In the CMS page editor click onto the link named **Advanced Settings** at the bottom of the popup window. As template, chose the default one, provided it contains at least one big placeholder.

Enter “*shop-order*” into the **Id**-field just below. This identifier is required by some templates which link directly onto the orders list view page. If this field is not set, some links onto this page might not work properly.

The Order Views must be rendered by their own CMS apphook. Locate the field **Application** and chose “*View Orders*”.

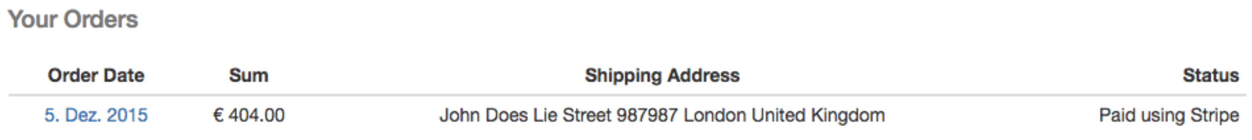
Below this “My Orders” page, add another page named “Thanks for Your Order”, “Danke für Ihre Bestellung” or “Gracias por su pedido”. Change into the **Advanced Settings** view and as the rendering template select “*Inherit the template of the nearest ancestor*”. Next enter “*shop-order-last*” into the **Id**-field just below. As **Application** chose again “*View Orders*”.

Add the Order list view via CMS-Cascade Plugin

Click onto **View on site** and change into front-end editing mode to use the grid-system of djangocms-cascade. Locate the main placeholder and add a **Row** followed by at least one **Column** plugin; both can be found in section **Bootstrap**. Below that column plugin, add a child named **Order Views** from section **Shop**.

We have to perform this operation a second time for the page named “Thanks for Your Order”. The context menus for copying and pasting may be helpful here.

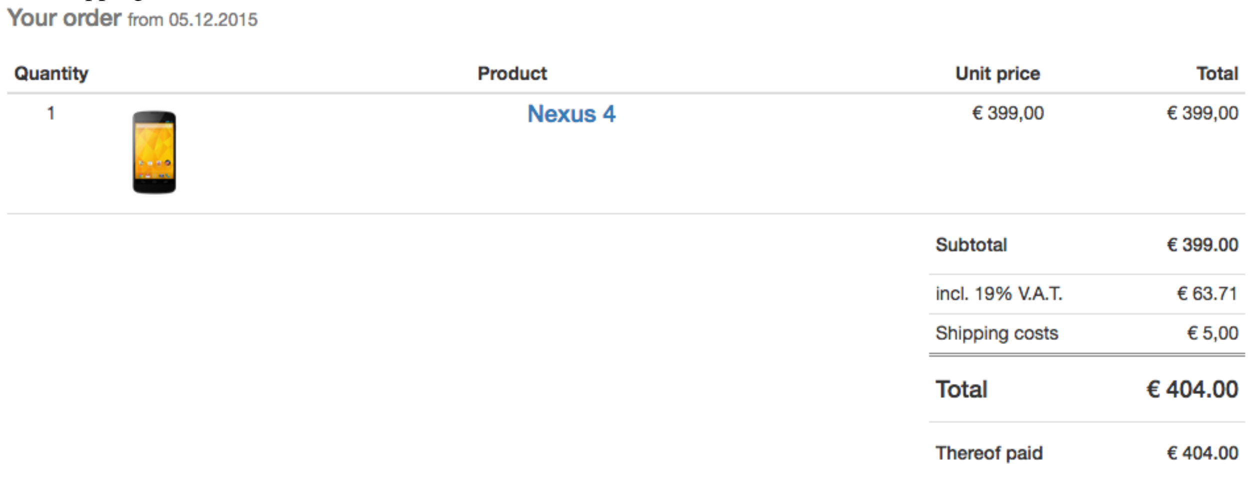
Note the the page “My Orders” handles two views: By invoking it as a normal CMS page, it renders a list of all orders the currently logged in customer has purchased at this shop:



Your Orders

Order Date	Sum	Shipping Address	Status
5. Dez. 2015	€ 404.00	John Does Lie Street 987987 London United Kingdom	Paid using Stripe

Clicking on one of the orders in this list, changes into a detail view, where one can see a list of items purchased during that shopping session:



Your order from 05.12.2015

Quantity	Product	Unit price	Total
1	Nexus 4	€ 399,00	€ 399,00

Subtotal	€ 399.00
incl. 19% V.A.T.	€ 63.71
Shipping costs	€ 5,00
Total	€ 404.00
Thereof paid	€ 404.00

The rendered list is a historical snapshot of the cart in the moment of purchase. If in the meantime the prices of products, tax rates, shipping costs or whatever changed, then that order object always keeps the values at that time in history. This even applies to translations. Strings are translated into their natural language on the moment of purchase. Therefore the labels added to the last rows of the cart, always are rendered in the language which was used during the checkout process.

Render templates

The path of the templates used to render the order views is constructed using the following rules:

- Look for a folder named according to the project's name, ie. `settings.SHOP_APP_LABEL` in lower case. If no such folder can be found, then use the folder named `shop`.
- Search for a subfolder named `order`.
- Search for a template named `list.html` or `detail.html`.

These templates are written to be easily extensible by the customized templates. To override them, add a template with the path, say `myshop/order/list.html` to the projects template folder.

4.10.3 Order Workflows

Order Workflows are simple plugins that allow the merchant to define rules in a programmatic way, which actions to perform, whenever a certain event happened. A typical event is the confirmation of a payment, which itself triggers further actions, say to print a delivery note.

Instead of implementing each possible combination for all of these use cases, the **djangoSHOP** framework offers a Finite State Machine, where only selected state transition can be marked as possible. These transition further can trigger other events themselves. This prevents to accidently perform invalid actions such as fulfilling orders, which haven't been paid yet.

In class `shop.models.order.BaseOrder` contains an attribute `status` which is of type `FSMField`. In practice this is a char-field, which can hold preconfigured states, but which *can not* be changed by program code. Instead, by calling specially decorated class methods, this state then changes from one or more allowed source states into one predefined target state. We denote this as a *state transition*.

An incomplete example:

```
class Order(models.Model):
    # other attributes

    @transition(field=status, source='new', target='created')
    def populate_from_cart(self, cart, request):
        # perform some side effects ...
```

Whenever an `Order` object is initialized, its `status` is *new* and not yet persisted in the database. As we have seen earlier, this object must be populated from the cart. If this succeeds, the `status` of our new `Order` object switches to *created*. This is the default state before proceeding to our payment providers.

In **djangoSHOP** the merchant can add as many payment providers he wants. This is done in `settings.py` through the configuration directive `SHOP_ORDER_WORKFLOWS` which takes a list of so called "*Order Workflow Mixin*" classes. On bootstrapping the application and constructing the `Order` class, it additionally inherits from these mixin classes. This gives the merchant an easy to configure, yet very powerful tool to model the selling process of his e-commerce site according to his needs. Say, we want to accept bank transfer in advance, so we must add `'shop.payment.defaults.PayInAdvanceWorkflowMixin'` to our configuration setting. Additionally we must assure that the checkout process has been configured to offer the corresponding cart modifier:

```
SHOP_CART_MODIFIERS = (
    ...
    'shop.modifiers.defaults.PayInAdvanceModifier',
    ...
)
```

This mixin class contains a few transition methods, lets for instance have a closer look onto

```
@transition(field='status', source=['created'], target='awaiting_payment')
def awaiting_payment(self):
     """Signals that an Order awaits payments."""
```

This method actually does nothing, beside changing the status from "*created*" to "*awaiting_payment*". It is invoked by the method `get_payment_request()` from `ForwardFundPayment`, which is the default payment provider of the configured `PayInAdvanceModifier` cart modifier.

The class `PayInAdvanceWorkflowMixin` has two other transition methods worth mentioning:

```
@transition(field='status', source=['awaiting_payment'],
    target='prepayment_deposited', conditions=[is_fully_paid],
    custom=dict(admin=True, button_name=_("Mark as Paid")))
def prepayment_fully_deposited(self):
    """Signals that the current Order received a payment."""
```

This method can be invoked by the Django admin backend when saving an existing Order object, but only under the condition that it is fully paid. The method `is_fully_paid()` iterates over all payments associated with its Order object, sums them up and compares them against the total. If the entered payment equals or exceeds the order's total, this method returns `True` and the condition for the given transition is met. This then adds a button labeled "*Mark as Paid*" at the bottom of the admin view. Whenever the merchant clicks on this button, the above method `prepayment_fully_deposited` is invoked. This then changes the order's status from "*awaiting_payment*" to "*prepayment_deposited*". The *Notifications* of **djangoSHOP** can intercept this transition change and perform preconfigured action, such as sending a payment confirmation email to the customer.

Now that the order has been paid, it time to fulfill it. For this a merchant can use the workflow mixin class `shop.shipping.defaults.CommissionGoodsWorkflowMixin`, which gives him a hand to keep track on the fulfillment of each order. Since this class doesn't know anything about an order status of "*prepayment_deposited*" (this is a private definition of the class `PayInAdvanceWorkflowMixin`), **djangoSHOP** provides a status to mark the payment of an order as confirmed. Therefore another transition is added to our mixin class, which is invoked automatically by the framework whenever the status changes to "*prepayment_deposited*":

```
@transition(field='status', source=['prepayment_deposited',
    'no_payment_required'], custom=dict(auto=True))
def acknowledge_prepayment(self):
    """Acknowledge the payment."""
    self.acknowledge_payment()
```

This status, "*payment_confirmed*", is known by all other workflow mixin classes and must be used as the source argument for their transition methods.

For further details on Finite State Machine transitions, please refer to the FSM docs. This however does not cover the contents of dictionary `custom`. One of the attributes in `custom` is `button="Any Label"` as explained in the FSM admin docs. The other is `auto=True` and has been introduced by **djangoSHOP** itself. It is used to automatically proceed from one target to another one, without manual intervention, such as clicking onto a button.

Signals

Each state transition emits a signal before and after performing the status change. These signals, `pre_transition` and `post_transition` can be received by any registered signal handler. In **djangoSHOP**, the notification framework listens for these events and creates appropriate notification e-mails, if configured.

But sometimes simple notifications are not enough, and the merchant's implementation must perform actions in a programmatic way. This for instance could be a query, which shall be sent to the goods management database, whenever a payment has been confirmed successfully.

In Django, we typically register signal handlers in the `ready` method of the merchant's application configuration:

Listing 4.8: myshop/apps.py

```
from django.apps import AppConfig

class MyShopConfig(AppConfig):
    name = 'my_shop'

    def ready(self):
        from django_fsm.signals import post_transition
        post_transition.connect(order_event_notification)

def order_event_notification(sender, instance=None, target=None, **kwargs):
    if target == 'payment_confirmed':
        # do whatever appropriate
```

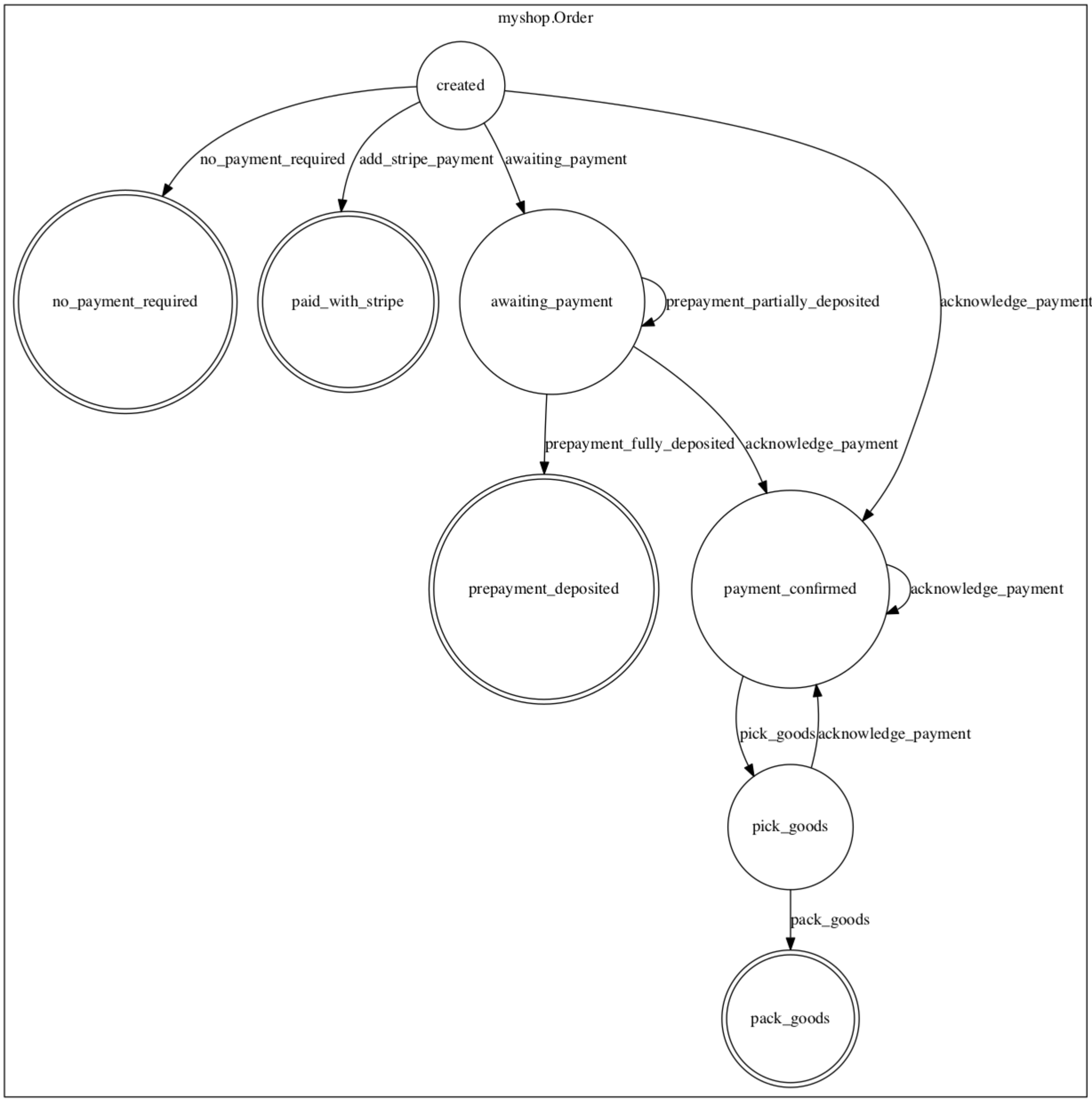
In the above order event notification, use `instance` to access the corresponding `Order` object.

Finite State Machine Diagram

If graphviz is installed on the operating system, it is pretty simple to render a graphical representation of the currently configured Finite State Machine. Simply invoke:

```
./manage.py ./manage.py graph_transitions -o fsm-graph.png
```

Applied to our demo shop, this gives the following graph:



4.10.4 Order Admin

The order editor likely is the most heavily used for each shop installation. Here the merchant must manage all incoming orders, payments, customer annotations, deliveries, etc. By automating common tasks, the backend shall prevent careless mistakes. For instance, it should be impossible to ship unpaid goods or to cancel a delivered order.

Since the **djangoSHOP** framework does not know which class model is used to implement an `Order`, it intentionally doesn't register its prepared administration class for that model. This has to be done by the project implementing the show. It allows to add additional fields and other mixin classes, before registration.

For instance, the admin class used to manage the `Order` model in our shop project, could be implemented as:

Listing 4.9: myshop/admin.py

```python
from django.contrib import admin
from shop.models.order import OrderModel
from shop.admin.order import (PrintOrderAdminMixin,
    BaseOrderAdmin, OrderPaymentInline, OrderItemInline)

@admin.register(OrderModel)
class OrderAdmin(PrintOrderAdminMixin, BaseOrderAdmin):
    fields = BaseOrderAdmin.fields + (
        ('shipping_address_text', 'billing_address_text',),)
    inlines = (OrderItemInline, OrderPaymentInline,)
```

The fields `shipping_address_text` and `billing_address_text` are not part of the abstract model class `BaseOrder` and therefore must be referenced separately.

Another useful mixin class to be added to this admin backend is `PrintOrderAdminMixin`. Whenever the status of an order is set to "*Pick the Goods*" a button labeled "*Print Delivery Note*" is added to the order admin form. Clicking on that button displays one ore more pages optimized for printing.

On the other hand, when the status of an order is set to "*Pack the Goods*" a button labeled "*Print Invoice*" is added to the order admin form.

The template for the invoice and delivery note can easily be adopted to the corporate design using plain HTML and CSS.

4.11 Managing the Deliver Process

Depending on the merchant's setup, an order can be considered as one inseparably unit, or if partial shipping shall be allowed, as a collection of single products, which can be delivered individually.

To enable partial shipping, assure the instantiation of both classes `shop.models.delivery.BaseDelivery` and `shop.models.delivery.BaseDeliveryItem`. The easiest way to do this is to import the materialized classes into an existing model class:

```python
from shop.models.defaults.delivery import Delivery, DeliveryItem
```

4.11.1 Partial Delivery Workflow

The class implementing the `Order`, requires additional methods provided by the mixin class `shop.shipping.delivery.PartialDeliveryWorkflowMixin`. Mix this class into the `Order` class by configuring

```python
SHOP_ORDER_WORKFLOWS = (
    # other workflow mixins
    'shop.shipping.defaults.PartialDeliveryWorkflowMixin',
)
```

Note: Do not combine this mixin with the class `CommissionGoodsWorkflowMixin`.

4.11.2 Administration Backend

To control partial delivery, add the class `shop.admin.delivery.DeliveryOrderAdminMixin` to the amin class class implementing an `Order`:

Listing 4.10: myshop/admin/order.py

```python
from django.contrib import admin
from shop.admin.order import BaseOrderAdmin
from shop.models.defaults.order import Order
from shop.admin.delivery import DeliveryOrderAdminMixin

@admin.register(Order)
class OrderAdmin(DeliveryOrderAdminMixin, BaseOrderAdmin):
    pass
```

4.11.3 Implementation Details

When partial delivery is activated, two additional tables are added to the database, one for each delivery and one for each delivered order item. This allows us to split up the quantity of in ordered item into two or more delivery objects. This can be useful, if a product is sold out, but the merchant wants to ship whatever is available on stock. He then creates a delivery object and assigns the available quantity to each linked delivery item.

If a product is not available at all anymore, the merchant can alternatively cancel that order item.

4.12 Designing an Address Model

Depending on the merchant's needs, the business model and the catchment area of the site, the used address models may vary widely. Since **djangoSHOP** allows to subclass almost every database model, addresses are no exception here. The class `shop.models.address.BaseAddress` only contains a foreign key to the Customer model and a priority field used to sort multiple addresses by relevance.

All the fields which make up an address, such as the addresse, the street, zip code, etc. are part of the concrete model implementing an address. It is the merchant's responsibility to define which address fields are required for his needs. Therefore the base address model does not contain any address related fields, they instead have to be declared by the merchant. A concrete implementation of the shipping address model may look like this:

..code-block:: python

from shop.models.address import BaseShippingAddress, ISO_3166_CODES

class ShippingAddress(BaseShippingAddress):

class Meta: verbose_name = "Shipping Address" verbose_name_plural = "Shipping Addresses"

addressee = models.CharField("Addressee", max_length=50) street = models.CharField("Street", max_length=50) zip_code = models.CharField("ZIP", max_length=10) location = models.CharField("Location", max_length=50) country = models.CharField("Country", max_length=3,

choices=ISO_3166_CODES)

Since the billing address may contain different fields, it must be defined separately from the shipping address. To avoid the duplicate definition of common fields for both models, use a mixin class such as:

..code-block:: python

from django.db import models from shop.models.address import BaseBillingAddress

class AddressModelMixin(models.Model): addressee = models.CharField(_("Addressee"), max_length=50) # other fields

class Meta: abstract = True

class BillingAddress(BaseBillingAddress, AddressModelMixin): tax_number = models.CharField("Tax number", max_length=50)

class Meta: verbose_name = "Billing Address" verbose_name_plural = "Billing Addresses"

4.12.1 Multiple Addresses

In **djangoSHOP**, if the merchant activates this feature, while setting up the site, customers can register more than one address. Multiple addresses can be activated, when editing the **Shipping Address Form Plugin** or the **Billing Address Form Plugin**.

Then during checkout, the customer can select one of a previously entered shipping- and billing addresses, or if he desires add a new one to his list of existing addresses.

4.12.2 How Addresses are used

Each active Cart object refers to one shipping address object and optionally one billing address object. This means that the customer can change those addresses whenever he uses the supplied address forms.

However, when the customer purchases the content of the cart, that address object is converted into a simple text string and stored inside the newly created Order object. This is to freeze the actual wording of the entered address. It also assures that the address used for delivery and printed on the invoice is immune against accidental changes after the purchasing operation.

4.12.3 Use Shipping Address for Billing

Most customers use their shipping address for billing. Therefore, unless you have really special needs, it is suggested to share all address fields required for shipping, also with the billing address. The customer then can reuse the shipping address for billing, if he desires to. Technically, if the billing address is unset, the shipping address is used anyway, but in **djangoSHOP** the merchant has to actively give permission to his customers, to reuse this address for billing.

The merchant has to actively allow this setting on the site, while editing the **Billing Address Form Plugin**.

4.12.4 Address Formatting

Whenever the customer fulfills the purchase operation, the corresponding shipping- and billing address objects are rendered into a short paragraph of plain text, separated by the newline character. This formatted address then is used to print address labels for parcel delivery and printed invoices.

It is the merchant's responsibility to format these addresses according to the local practice. A customized address template must be added into the merchant's implementation below the `templates` folder named `myshop/shipping_address.txt` or `myshop/billing_address.txt`. If both address models share the same fields, we may also use `myshop/address.txt` as a fallback. Such an address template may look like:

Listing 4.11: myshop/address.txt

```
{{ address.addressee }}{% if address.supplement %}
{{ address.supplement }}{% endif %}
{{ address.street }}
{{ address.zip_code }} {{ address.location }}
{{ address.get_country_display }}
```

This template is used by the method `as_text()` as found in each address model.

4.12.5 Address Forms

The address form, where customers can insert their address, is generated automatically and in a DRY manner. This means that whenever a field is added, modified or removed from the address model, the corresponding fields in the address input form, reflect those changes without manual intervention. When creating the form template, we have to write it using the `as_div()` method. This method also adds automatic client-side form validation to the corresponding HTML code.

Address Form Styling

One problem which remains with automatic form generation, is how to style the input fields. Therefore, **djangoSHOP** wraps every input field into a `<div>`-element using a CSS class named according to the field. This for instance is useful to shorten some input fields and/or place it onto the same line.

Say, any of our address forms contain the fields `zip_code` and `location` as shown in the example above. Then they may be styled as

```
.shop-address-zip_code {
    width: 35%;
    display: inline-block;
}

.shop-address-location {
    width: 65%;
    display: inline-block;
    margin-left: -4px;
    padding-left: 15px;
}
```

so that the ZIP field is narrower and precedes the location field on the same line.

4.13 Full Text Search

How should a customer find the product he desires in a more or less unstructured collection of countless products. Hierarchical navigation often doesn't work and takes too much time. Thanks to the way we use the Internet today, most site visitors expect one central search field in the main navigation bar of a site.

4.13.1 Search Engine API

In Django the most popular API for full-text search is Haystack. While other indexing backends, such as Solr and Whoosh might work as well, the best results have been achieved with Elasticsearch. Therefore this documentation focuses exclusively on Elasticsearch. And since in **djangoSHOP** every programming interface uses REST, search is no exception here. Fortunately there is a project named drf-haystack, which "restifies" our search results, if use use special serializers.

In this document we assume that the merchant only wants to index his products, but not any arbitrary content, such as for example the terms and condition, as found outside **djangoSHOP**, but inside **djangoCMS**.

Configuration

Install the Elasticsearch binary. Currently Haystack only supports versions smaller than 2. Then start the service in daemon mode:

```
./path/to/elasticsearch-version/bin/elasticsearch -d
```

Check if the server answers on HTTP requests. Pointing a browser onto port http://localhost:9200/ should return something similar to this:

```
$ curl http://localhost:9200/
{
  "status" : 200,
  "name" : "Ape-X",
  "cluster_name" : "elasticsearch",
  "version" : {
    ...
  },
}
```

In `settings.py`, check that `'haystack'` has been added to `INSTALLED_APPS` and connects the application server with the Elasticsearch database:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.elasticsearch_backend.ElasticsearchSearchEngine',
        'URL': 'http://localhost:9200/',
        'INDEX_NAME': 'myshop-default',
    },
}
```

In case we need indices for different natural languages on our site, we shall add the non-default languages to this Python dictionary using a different `INDEX_NAME` for each of them.

Finally configure the site, so that search queries are routed to the correct index using the currently active natural language:

```
HAYSTACK_ROUTERS = ('shop.search.routers.LanguageRouter',)
```

4.13.2 Indexing the Products

Before we start to search for something, we first must populate its indices. In Haystack one can create more than one kind of index for each item being added to the search database.

Each product type requires its individual indexing class. Note that Haystack does some autodiscovery, therefore this class must be added to a file named `search_indexex.py`. For our product model `SmartCard`, this indexing class then may look like:

Listing 4.12: myshop/search_indexes.py

```
from shop.search.indexes import ProductIndex
from haystack import indexes

class SmartCardIndex(ProductIndex, indexes.Indexable):
    catalog_media = indexes.CharField(stored=True,
        indexed=False, null=True)
    search_media = indexes.CharField(stored=True,
        indexed=False, null=True)

    def get_model(self):
        return SmartCard

    # more methods ...
```

While building the index, Haystack performs some preparatory steps:

Populate the reverse index database

The base class for our search index declares two fields for holding the reverse indexes and a few additional fields to store information about the indexed product entity:

Listing 4.13: shop/indexes.py

```
class ProductIndex(indexes.SearchIndex):
    text = indexes.CharField(document=True,
        indexed=True, use_template=True)
    autocomplete = indexes.EdgeNgramField(indexed=True,
        use_template=True)

    product_name = indexes.CharField(stored=True,
        indexed=False, model_attr='product_name')
    product_url = indexes.CharField(stored=True,
        indexed=False, model_attr='get_absolute_url')
```

The first two index fields require a template which renders plain text, which is used to build a reverse index in the search database. The `indexes.CharField` is used for a classic reverse text index, whereas the `indexes.EdgeNgramField` is used for autocompletion.

Each of these index fields require their own template. They *must* be named according to the following rules:

```
search/indexes/myshop/<product-type>_text.txt
```

and

```
search/indexes/myshop/<product-type>_autocomplete.txt
```

and be located inside the project's template folder. The `<product-type>` is the classname in lowercase of the given product model. Create two individual templates for each product type, one for text search and one for autocompletion.

An example:

Listing 4.14: search/indexes/smartcard_text.txt

```
{{ object.product_name }}
{{ object.product_code }}
{{ object.manufacturer }}
{{ object.description|striptags }}
{% for page in object.cms_pages.all %}
{{ page.get_title }}{% endfor %}
```

The last two fields are used to store information about the product's content, side by side with the indexed entities. That's a huge performance booster, since this information otherwise would have to be fetched from the relational database, item by item, and then being rendered while preparing the search query result.

We can also add fields to our index class, which stores pre-rendered HTML. In the above example, this is done by the fields `catalog_media` and `search_media`. Since we do not provide a model attribute, we must provide two methods, which creates this content:

Listing 4.15: myshop/search_indexes.py

```
class SmartCardIndex(ProductIndex, indexes.Indexable):
    # other fields and methods ...

    def prepare_catalog_media(self, product):
        return self.render_html('catalog', product, 'media')

    def prepare_search_media(self, product):
        return self.render_html('search', product, 'media')
```

These methods themselves invoke `render_html` which takes the product and renders it using a templates named `catalog-product-media.html` or `search-product-media.html` respectively. These templates are looked for in the folder `myshop/products` or, if not found there in the folder `shop/products`. The HTML snippets for catalog-media are used for autocompletion search, whereas search-media is used for normal a normal full-text search invocation.

Building the Index

To build the index in Elasticsearch, invoke:

```
./manage.py rebuild_index --noinput
```

Depending on the number of products in the database, this may take some time.

4.13.3 Search Serializers

Haystack for Django REST Framework is a small library aiming to simplify using Haystack with Django REST Framework. It takes the search results returned by Haystack, treating them the similar to Django database models when serializing their fields. The serializer used to render the content for this demo site, may look like:

Listing 4.16: myshop/serializers.py

```
from rest_framework import serializers
from shop.search.serializers import ProductSearchSerializer as ProductSearchSerializerBase
from .search_indexes import SmartCardIndex, SmartPhoneIndex

class ProductSearchSerializer(ProductSearchSerializerBase):
    media = serializers.SerializerMethodField()

    class Meta(ProductSearchSerializerBase.Meta):
        fields = ProductSearchSerializerBase.Meta.fields + ('media',)
        index_classes = (SmartCardIndex, SmartPhoneIndex)

    def get_media(self, search_result):
        return search_result.search_media
```

This serializer is part of the project, since we must adopt it to whatever content we want to display on our site, whenever a visitor enters some text into the search field.

4.13.4 Search View

In the Search View we link the serializer together with a djangoCMS apphook. This `ProductSearchApp` can be added to the same file, we already used to declare the `ProductsListApp` used to render the catalog view:

Listing 4.17: myshop/cms_app.py

```python
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool

class ProductSearchApp(CMSApp):
    name = _("Search")
    urls = ['myshop.urls.search']

apphook_pool.register(ProductSearchApp)
```

as all apphooks, it requires a file defining its urlpatterns:

Listing 4.18: myshop/urls/search.py

```python
from django.conf.urls import patterns, url
from shop.search.views import SearchView
from myshop.serializers import ProductSearchSerializer

urlpatterns = patterns('',
    url(r'^', SearchView.as_view(
        serializer_class=ProductSearchSerializer,
    )),
)
```

Search Results

As with all other pages in **djangoSHOP**, the page displaying our search results is a normal CMS page too. It is suggested to create this page on the root level of the page tree.

As the page title use "*Search*" or whatever is appropriate in our natural language. Then we change into advanced setting.

As a template use one with a big placeholder, since it must display our search results.

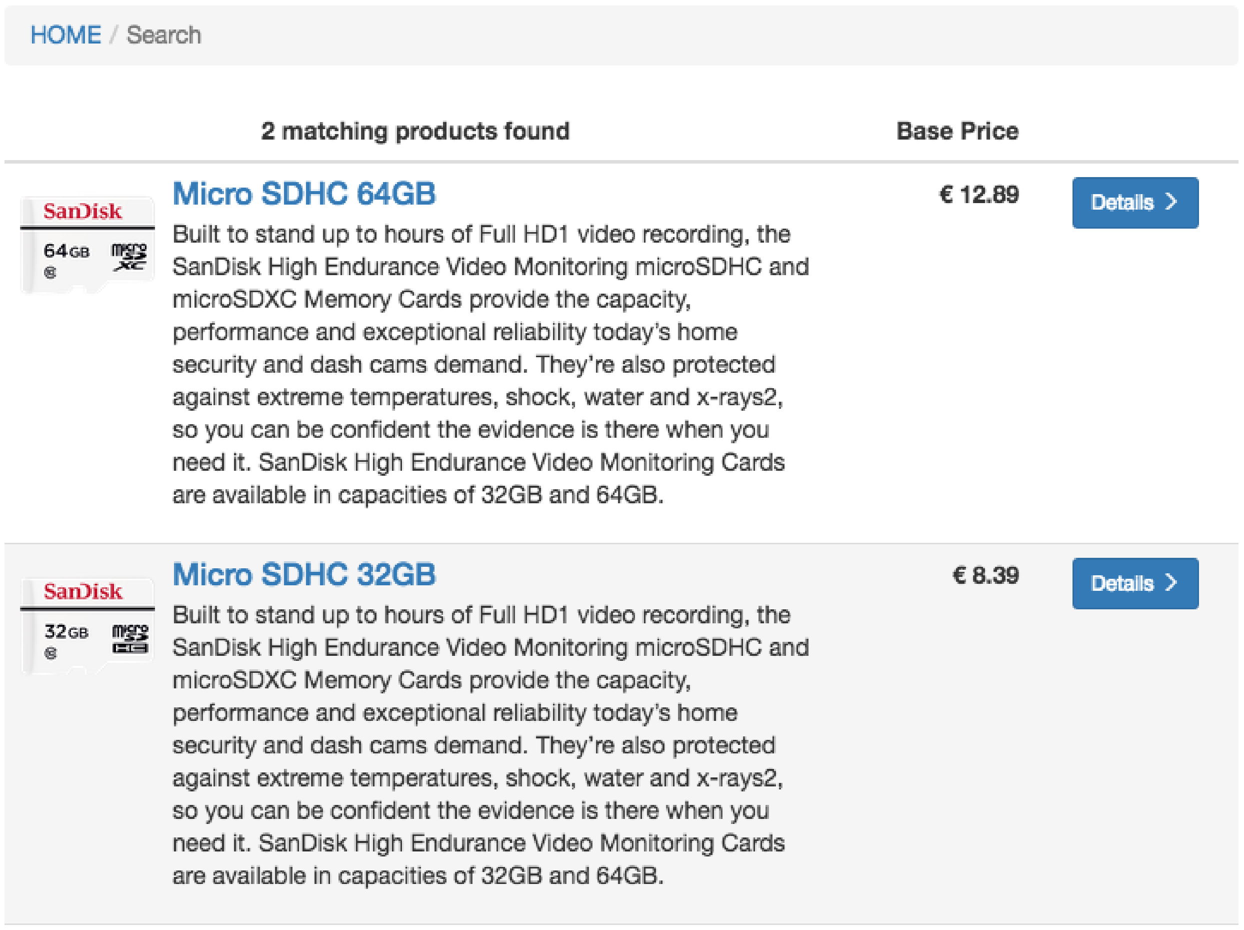
In the page **Id** field, use "shop-search-product". Some prepared default templates use this hard coded string.

Set the input field **Soft root** to checked. This hides this special page from our menu list.

As **Application**, select "*Search*". This selects the apphook we created in the previous section.

Then save the page, change into **Structure** mode and locate the Main Content Container. Add a container with a Row and Column. As the child of this column chose the **Search Results** plugin from section **Shop**.

Finally publish the page and enter some text into the search field. It should render a list of found products.


HOME / Search

2 matching products found		Base Price	
	Micro SDHC 64GB Built to stand up to hours of Full HD1 video recording, the SanDisk High Endurance Video Monitoring microSDHC and microSDXC Memory Cards provide the capacity, performance and exceptional reliability today's home security and dash cams demand. They're also protected against extreme temperatures, shock, water and x-rays2, so you can be confident the evidence is there when you need it. SanDisk High Endurance Video Monitoring Cards are available in capacities of 32GB and 64GB.	€ 12.89	Details >
	Micro SDHC 32GB Built to stand up to hours of Full HD1 video recording, the SanDisk High Endurance Video Monitoring microSDHC and microSDXC Memory Cards provide the capacity, performance and exceptional reliability today's home security and dash cams demand. They're also protected against extreme temperatures, shock, water and x-rays2, so you can be confident the evidence is there when you need it. SanDisk High Endurance Video Monitoring Cards are available in capacities of 32GB and 64GB.	€ 8.39	Details >

4.13.5 Autocompletion in Catalog List View

As we have seen in the previous example, the Product Search View is suitable to search for any item in the product database. However, the site visitor sometimes might just refine the list of items shown in the catalog's list view. Here loading a new page which uses a completely different layout, may by inappropriate.

Instead, when someone enters some text into the search field, **djangoSHOP** starts to narrow down the list of items in the Catalog List View by typing query terms into the search field. This is specially useful in situations where hundreds of products are displayed together on the same page and the customer needs to pick out the correct one by entering some search terms.

To extend the existing Catalog List View for autocompletion, locate the file containing the urlpatterns, which are used by the apphook `ProductsListApp`. In doubt, consult the file `myshop/cms_app.py`.

Into these urlpatterns add the following entry:

```
from django.conf.urls import patterns, url
from shop.search.views import SearchView
from myshop.serializers import CatalogSearchSerializer

urlpatterns = patterns('',
    # previous patterns
    url(r'^search-catalog$', SearchView.as_view(
        serializer_class=CatalogSearchSerializer,
    )),
    # other patterns
)
```

Note: Be careful the the regular expression for `^search-catalog$` matches before the product's detail view, which usually is looks for patterns matching `^(?P<slug>[\w-]+)$`.

The `CatalogSearchSerializer` used here is very similar to the `ProductSearchSerializer` we have seen in the previous section. The only difference is, that instead of the `search_media` field is uses the `catalog_media` field, which renders the result items media in a layout appropriate for the catalog's list view.

4.14 Notifications

Whenever the status in model `Order` changes, the built-in Finite State Machine emits a signal using Django's signaling framework. These signals are received by **djangoSHOP**'s Notification Framework.

4.14.1 Notification Admin

In Django's admin backend on **Start > Shop > Notification**, the merchant can configure which email to send to whom, depending on each of the emitted events. When adding or editing a notification, we get a form mask with four input fields:

Notification Identifier

An arbitrary name used to distinguish the different notifications. Its up to the merchant to chose a meaningful name, "Order confirmed, paid with PayPal" could for instance be a good choice.

Event Type

Each *Order Workflows* declares a set of transition targets. For instance, the class `PayInAdvanceWorkflowMixin` declares these targets: "*Awaiting a forward fund payment*", "*Prepayment deposited*" and "*No Payment Required*".

The merchant can attach a notification for each of these transition targets. Here he must chose one from the prepared collection.

The Recipient

Transitions events are transmitted for changes in the order status. Each order belongs to one customer, and normally he's the first one to be informed, if something changes.

But other persons in the context of this e-commerce site might also be interested into a notification. In **djangoSHOP** all staff Users qualify, as it is assumed that they belong to the group eligible to manage the site.

Email Templates

From the section **Start > Post Office > Email Templates**, chose on of the *Templates for Emails*.

Notification attachments

Chose none, one or more static files to be attached to each email. This typically is a PDF with the terms and conditions. We normally want to send them only to our customers, but not to the staff users, otherwise we'd fill up their mail inbox with countless attachments.

4.14.2 Post Office

Emails for order confirmations are send asynchronously by **djangoSHOP**. The reason for this is that it sometimes takes a few seconds for an application server to connect via SMTP, and deliver an Email. It is unacceptable to do this synchronously during the most sensitive phase of a purchase operation.

Therefore **djangoSHOP** sends all generated emails using the queuing mail system Post Office. This app can hold a set of different email templates, which use the same template language as Django itself. Emails can be rendered using plain text, HTML or both.

When emails are queued, the chosen template object is stored side by side with its context serialized as JSON. These queued emails are accessible in Django's admin backend at **Start > Post Office > Emails**. Their status can either be "*queued*", "*sent*" or "*failed*".

As an offline operation, `./manage.py send_queued_mail` renders and sends queued emails to the given recipient. During this step, the given template is rendered applying the stored context. Their status then changes to "*sent*", or in case of a problem to "*failed*".

If **djangoSHOP** is configured to run in a multilingual environment, post office renders the email in the language used during order creation.

Templates for Emails

The **Message** fields can contain any code, which is valid for Django templates. Frequently, a summary of the order is rendered in these emails, creating a list of ordered items. This list often is common across all email templates, and therefore it is recommended to prepare it in a base template for being reused. In the merchants project folder, create those base email templates inside the folder `templates/myshop/email/...`. Then inside the **Message** fields, these templates can be loaded and expanded using the well known templatetag

```
{% extends "myshop/email/somebase.html" %}
```

Caveats when using an HTML Message

Displaying HTML in email clients is a pain. Nobody really can say, which HTML tags are allowed in which client – and there are many email readers out there, far more than Internet browsers.

Therefore when designing HTML templates for emails, one must be really, really conservative. It may seem anachronistic, but still a best practice is to use the `<table>` element, and if necessary, nest it into their `<td>` (tables data) elements. Moreover, use inline styles rather than a `<style>` element containing blocks of CSS. It is recommended to use a special email framework to avoid nasty quirks, when rendering the templates.

Images can be embedded into HTML emails using two different methods. One is to host the image on the web-server and to build an absolute URI referring it. Therefore **djangoSHOP** enriches the object `RenderContext` with the base URI for that web-site and stores it as context variable named `ABSOLUTE_BASE_URI`. For privacy reasons, most email clients do not load externally hosted images by default – the customer then must actively request to load them from the external sources.

Another method for adding images to HTML emails is to inline their payload. This means that images, instead of referring them by URI, are inlined as a base64-encoded string. Easy-thumbnails offers a template filter named `data_uri` to perform this operation. This of course blows up the overall size of an email and shall only be used for small an medium sized images.

4.15 REST Serializers

God application programming style is to strictly separate of *Models*, *Views* and *Controllers*. In typical classic Django jargon, *Views* act as, what outsiders normally would denote a controller.

Controllers can sometimes be found on the server and sometimes on the client. In **djangoSHOP** a significant portion of the controller code is written in JavaScript in the form of Angular directives.

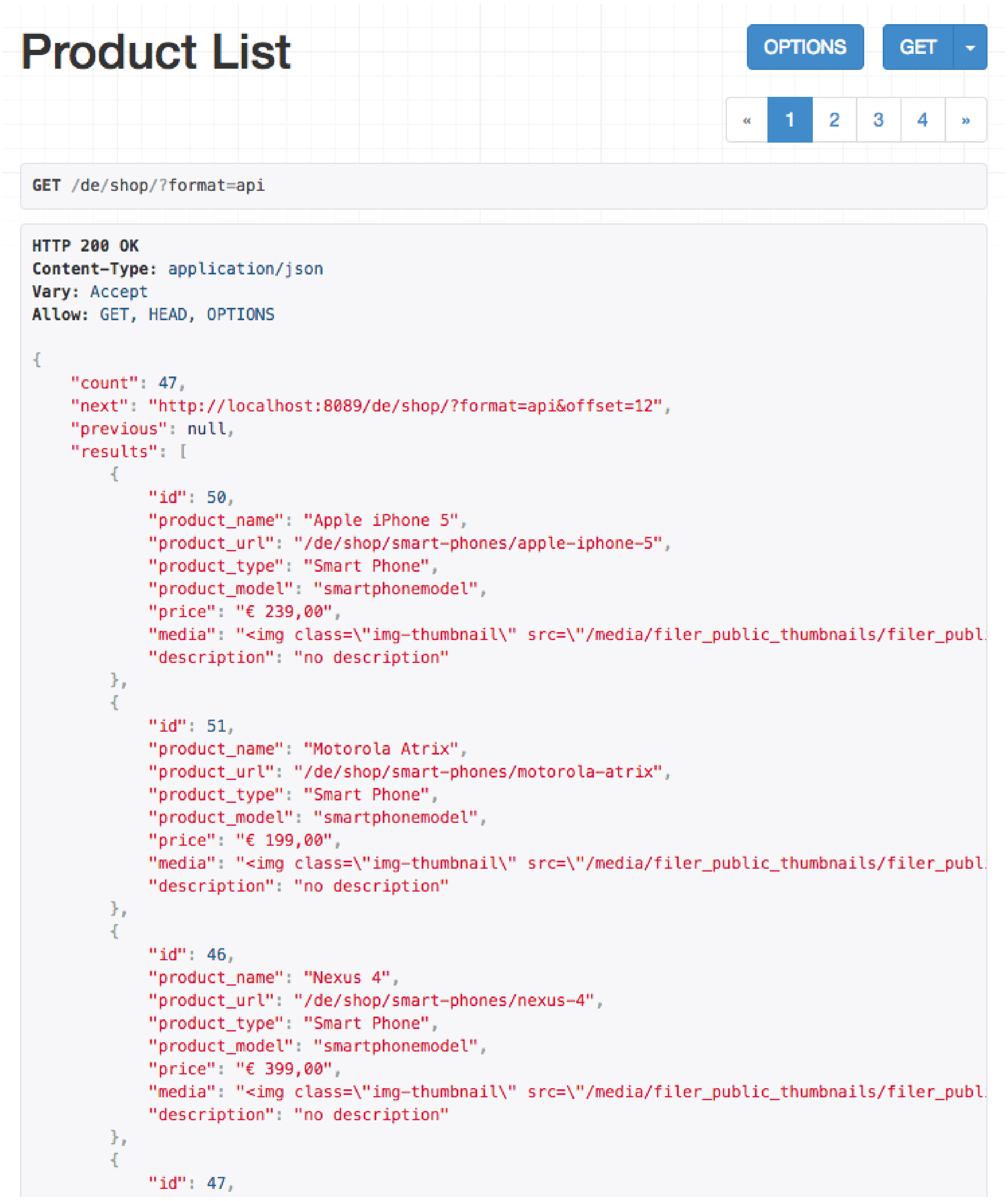
Therefore, all data exchange between the *View* and the *Model* must be performed in a serializable format, namely JSON. This allows us to use the same business logic for the server, as well as for the client. It also means, that we could create native mobile apps, which communicate with a web-application, without ever seeing a line of HTML code.

4.15.1 Every URL is a REST endpoint

Every URL which is part of part of **djangoSHOP**, namely the product's list and detail views, the cart and checkout views, the order list and detail views, they all are REST endpoints. What does that mean?

Catalog List View

Say, we are working with the provided demo shop, then the product's list view is available at http://localhost:8000/de/shop/ . By appending `?format=json` to the URL, the raw data making up our product list, is rendered as a JSON object. For humans, this is difficult to read, therefore the Django Restframework offers a version which is more legible: Instead of the above, we invoke the URL as http://localhost:8000/de/shop/?format=api . This renders the list of products as:

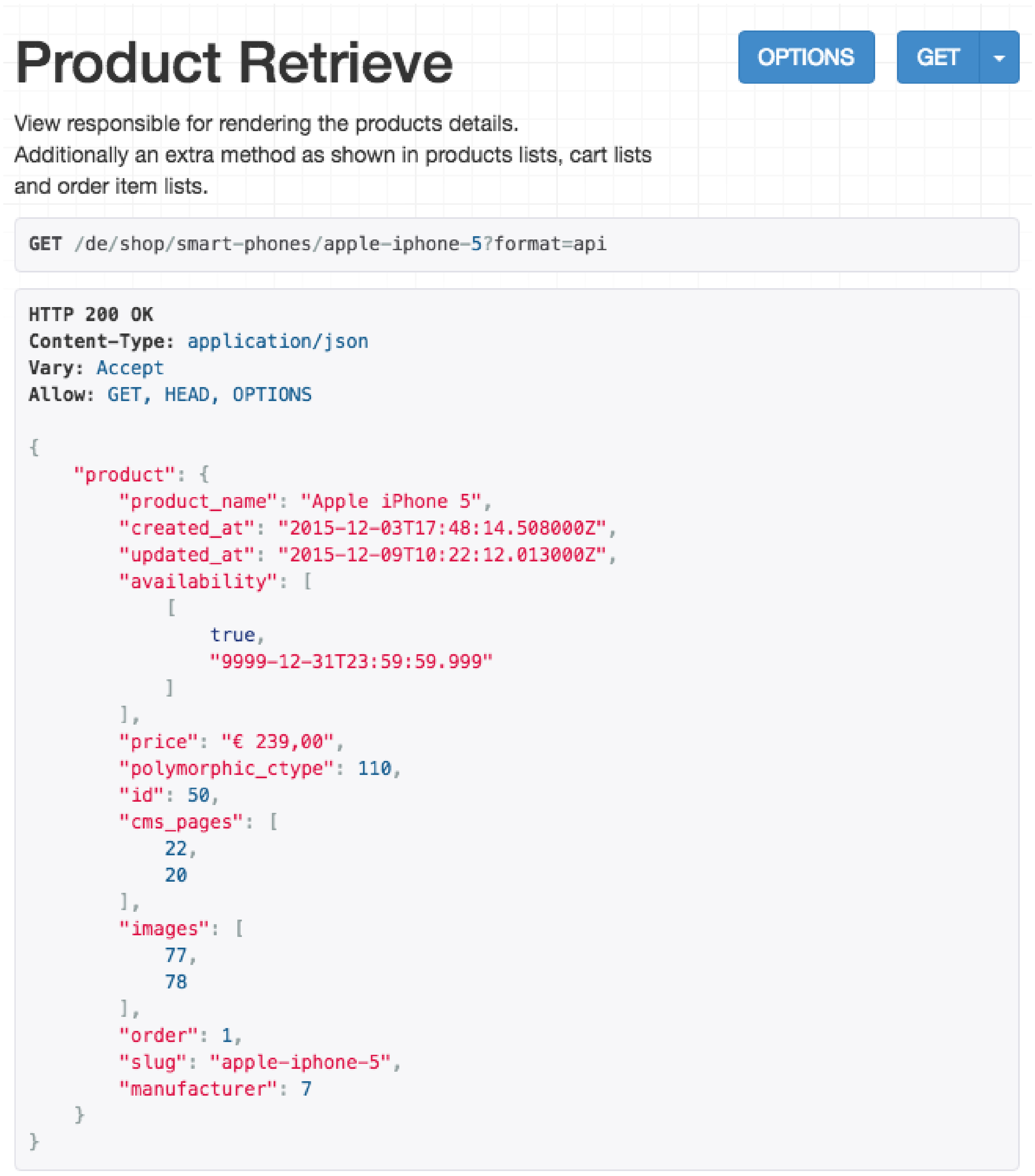


```
GET /de/shop/?format=api
```

```
HTTP 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "count": 47,
    "next": "http://localhost:8089/de/shop/?format=api&offset=12",
    "previous": null,
    "results": [
        {
            "id": 50,
            "product_name": "Apple iPhone 5",
            "product_url": "/de/shop/smart-phones/apple-iphone-5",
            "product_type": "Smart Phone",
            "product_model": "smartphonemodel",
            "price": "€ 239,00",
            "media": "<img class=\"img-thumbnail\" src=\"/media/filer_public_thumbnails/filer_publ
            "description": "no description"
        },
        {
            "id": 51,
            "product_name": "Motorola Atrix",
            "product_url": "/de/shop/smart-phones/motorola-atrix",
            "product_type": "Smart Phone",
            "product_model": "smartphonemodel",
            "price": "€ 199,00",
            "media": "<img class=\"img-thumbnail\" src=\"/media/filer_public_thumbnails/filer_publ
            "description": "no description"
        },
        {
            "id": 46,
            "product_name": "Nexus 4",
            "product_url": "/de/shop/smart-phones/nexus-4",
            "product_type": "Smart Phone",
            "product_model": "smartphonemodel",
            "price": "€ 399,00",
            "media": "<img class=\"img-thumbnail\" src=\"/media/filer_public_thumbnails/filer_publ
            "description": "no description"
        },
        {
            "id": 47,
```

Catalog Detail View

By following a URL of a product's detail view, say http://localhost:8000/de/shop/smart-phones/apple-iphone-5?format=api , one may check the legible representation such as:



Product Retrieve

View responsible for rendering the products details.
Additionally an extra method as shown in products lists, cart lists and order item lists.

```
GET /de/shop/smart-phones/apple-iphone-5?format=api
```

```
HTTP 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "product": {
        "product_name": "Apple iPhone 5",
        "created_at": "2015-12-03T17:48:14.508000Z",
        "updated_at": "2015-12-09T10:22:12.013000Z",
        "availability": [
            [
                true,
                "9999-12-31T23:59:59.999"
            ]
        ],
        "price": "€ 239,00",
        "polymorphic_ctype": 110,
        "id": 50,
        "cms_pages": [
            22,
            20
        ],
        "images": [
            77,
            78
        ],
        "order": 1,
        "slug": "apple-iphone-5",
        "manufacturer": 7
    }
}
```

Routing to these endpoints

Since we are using CMS pages to display the catalog's list view, we must provide an apphook to attach it to this page. These catalog apphooks are not part of the shop framework, but must be created and added to the project:

Listing 4.19: myshop/cms_app.py

```
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool

class CatalogListApp(CMSApp):
    name = "Catalog List"
    urls = ['myshop.urls.catalog']

apphook_pool.register(CatalogListApp)
```

We now must add routes for all sub-URLs of the given CMS page implementing the catalog list:

Listing 4.20: myshop/urls/catalog.py

```
from django.conf.urls import patterns, url
from rest_framework.settings import api_settings
from shop.rest.filters import CMSPagesFilterBackend
from shop.views.catalog import (AddToCartView, CMSPageProductListView,
    ProductRetrieveView)
from myshop.serializers import (ProductSummarySerializer,
    ProductDetailSerializer)

urlpatterns = patterns('',
    url(r'^$', CMSPageProductListView.as_view(
        serializer_class=ProductSummarySerializer,
    )),
    url(r'^(?P<slug>[\w-]+)$', ProductRetrieveView.as_view(
        serializer_class=ProductDetailSerializer,
    )),
    url(r'^(?P<slug>[\w-]+)/add-to-cart', AddToCartView.as_view()
    ),
)
```

Products List View

The urlpattern matching the regular expression `^$` routes onto the catalog list view class `shop.views.catalog.CMSPageProductListView` passing in a special serializer class, for example `myshop.serializers.ProductSummarySerializer`. This has been customized to represent our product models in our catalog templates. Since the serialized data now is available as a Python dictionary or as a plain Javascript object, these templates then can be rendered by the Django template engine, as well as by the client using for instance AngularJS.

This View class, which inherits from `rest_framework.generics.ListAPIView` accepts a list of filters for restricting the list of items.

As we (ab)use CMS pages as categories, we somehow must assign them to our products. Therefore our example project assigns a many-to-many field named `cms_pages` to our Product model. Using this field, the merchant can assign each product to one or more CMS pages, using the apphook `Products List`.

This special `filter_backend`, `shop.rest.filters.CMSPagesFilterBackend`, is responsible for restricting selected products on the current catalog list view.

Product Detail View

The urlpattern matching the regular expression `^(?P<slug>[\w-]+)$` routes onto the class `shop.views.catalog.ProductRetrieveView` passing in a special serializer class, `myshop.serializers.ProductDetailSerializer` which has been customized to represent our product model details.

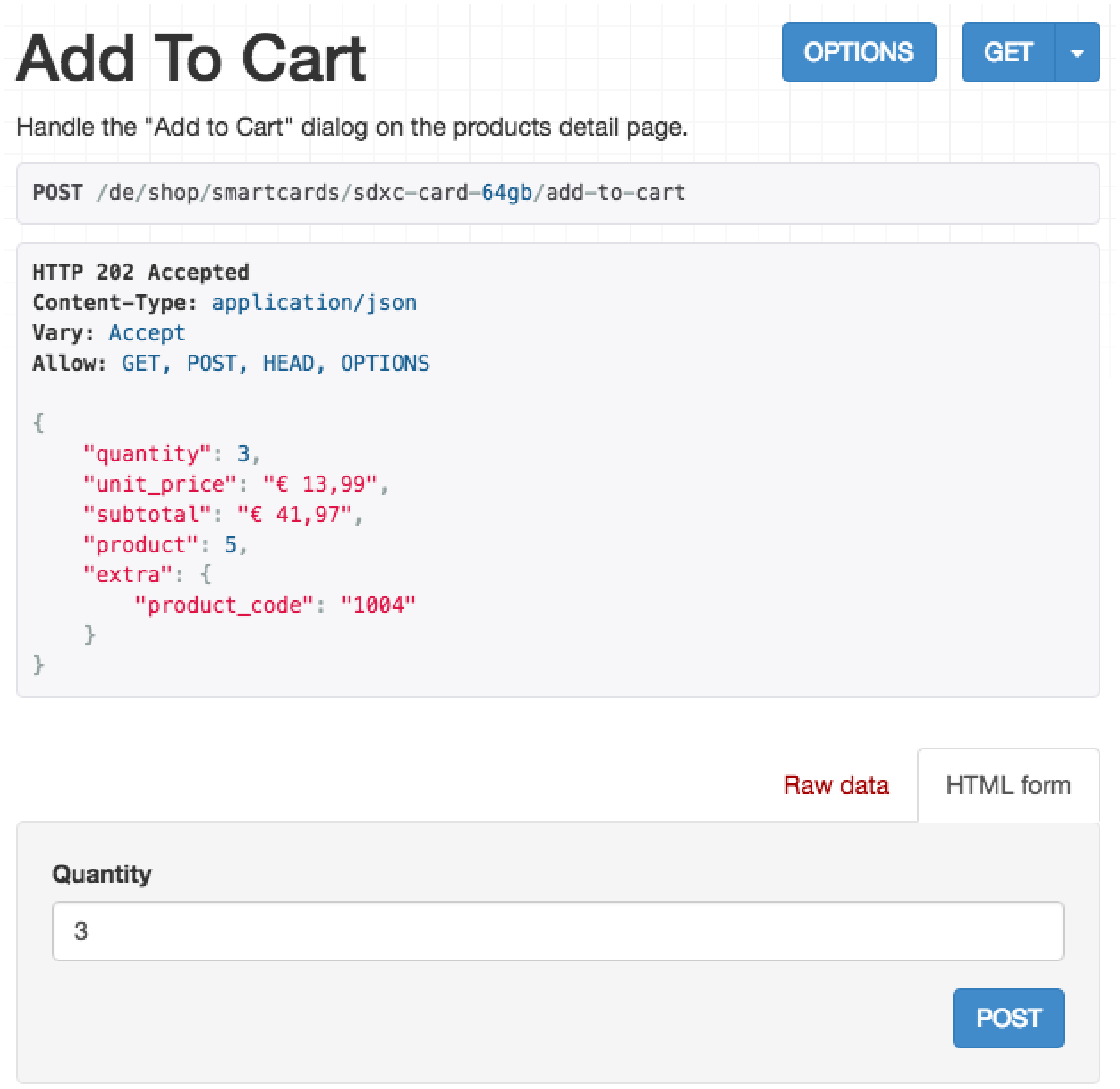
This View class inherits from `rest_framework.generics.RetrieveAPIView`. In addition to the given `serializer_class` it can accept these fields:

- `lookup_field`: Model field to look up for the retrieved product. This defaults to `slug`.
- `lookup_url_kwarg`: URL argument as used by the matching RegEx. This defaults to `slug`.
- `product_model`: Restrict to products of this type. Defaults to `ProductModel`.

Add Product to Cart

The product detail view requires another serializer, the so called `AddToCartSerializer`. This serializer is responsible for controlling the number of items being added to the cart and gives feedback on the subtotal of that potential cart item.

By appending the special string `add-to-cart` to the URL of a product's detail view, say http://localhost:8000/de/shop/smart-phones/apple-iphone-5/add-to-cart?format=api , one may check the legible representation of this serializer:



Handle the "Add to Cart" dialog on the products detail page.

```
POST /de/shop/smartcards/sdxc-card-64gb/add-to-cart
```

```
HTTP 202 Accepted
Content-Type: application/json
Vary: Accept
Allow: GET, POST, HEAD, OPTIONS

{
    "quantity": 3,
    "unit_price": "€ 13,99",
    "subtotal": "€ 41,97",
    "product": 5,
    "extra": {
        "product_code": "1004"
    }
}
```

Quantity

3

This serializer is slightly different than the previous ones, because it not only serializes data and sends it from the server to the client, but it also deserializes data submitted from the client back to the server using a post-request. This normally is the quantity, but in more elaborated use cases, it also could contain attributes to distinguish product variations. The `AddSmartPhoneToCartSerializer` for example, uses this pattern.

Since we may create our own *Add this Product to Cart Serializer* for each product type in our shop, hence overriding its functionality with a customized implementation, such a serializer may return any other information relevant to the customer. This could for instance be a rebate or just an update of the availability.

Cart and Checkout Views

CMS pages containing forms to edit the cart and the checkout views, do not require any URL routing, because their HTML is rendered by the CMS plugin system, whereas form submissions are handled by hard coded REST endpoints. These URLs are exclusively used by Ajax requests and never visible in the URL line of our browser. Those endpoints are configured by adding them to the root resolver at a project level:

Listing 4.21: myshop/urls.py

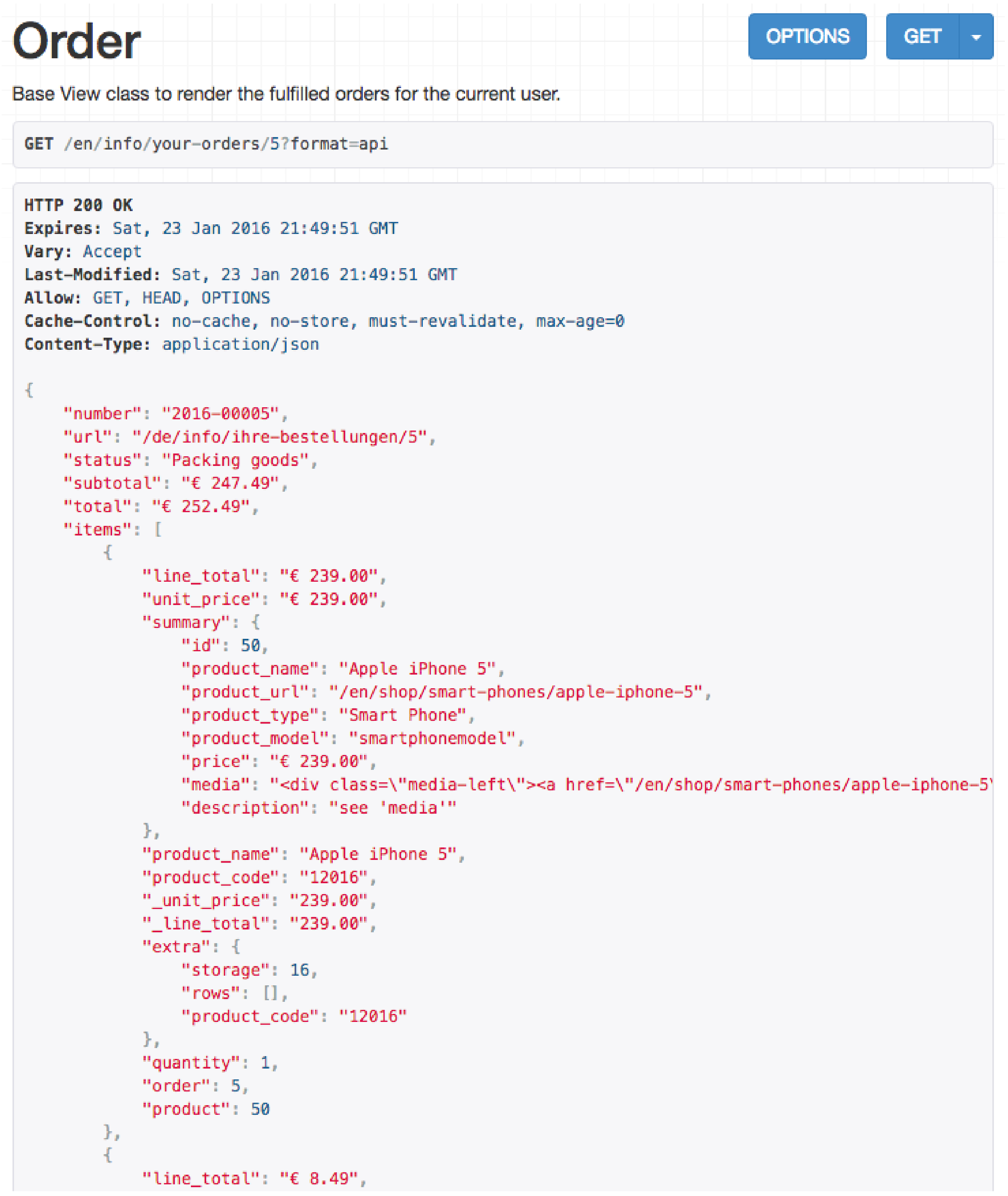
```
urlpatterns = patterns('',
    ...
    url(r'^shop/', include('shop.urls', namespace='shop')),
    ...
)
```

The serializers of the cart then can be accessed at http://localhost:8000/shop/api/cart/ , those of the watch-list at http://localhost:8000/shop/api/watch/ and those handling the various checkout forms at http://localhost:8000/shop/api/checkout/ . Accessing these URLs can be useful, specially when debugging JavaScript code.

Order List and Detail Views

The Order List and Detail Views must be accessible through a CMS page, therefore we need a speaking URL. This is similar to the Catalog List View. This means that the Order Views require the apphook named "*View Orders*", which must be configured in the advanced settings of the Order's CMS pages. This apphook is shipped with **djangoSHOP** itself and can be found at `shop/cms_app.py`.

As with all other Views used by **djangoSHOP**, the content of this View can also be rendered in its dictionary structure, instead of HTML. Just append `?format=api` to the URL and get the Order details. In our myshop example this may look like:



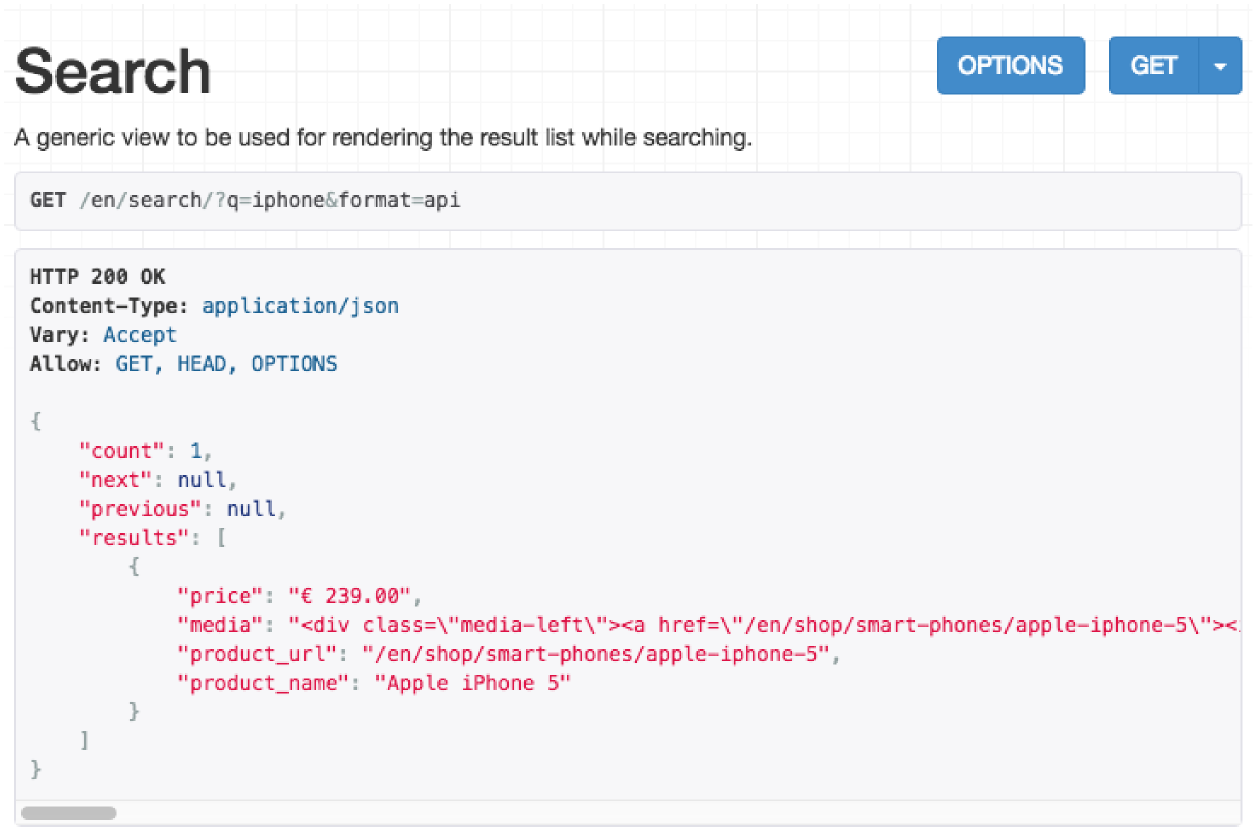
Base View class to render the fulfilled orders for the current user.

```
GET /en/info/your-orders/5?format=api
```

```
HTTP 200 OK
Expires: Sat, 23 Jan 2016 21:49:51 GMT
Vary: Accept
Last-Modified: Sat, 23 Jan 2016 21:49:51 GMT
Allow: GET, HEAD, OPTIONS
Cache-Control: no-cache, no-store, must-revalidate, max-age=0
Content-Type: application/json

{
    "number": "2016-00005",
    "url": "/de/info/ihre-bestellungen/5",
    "status": "Packing goods",
    "subtotal": "€ 247.49",
    "total": "€ 252.49",
    "items": [
        {
            "line_total": "€ 239.00",
            "unit_price": "€ 239.00",
            "summary": {
                "id": 50,
                "product_name": "Apple iPhone 5",
                "product_url": "/en/shop/smart-phones/apple-iphone-5",
                "product_type": "Smart Phone",
                "product_model": "smartphonemodel",
                "price": "€ 239.00",
                "media": "<div class=\"media-left\"><a href=\"/en/shop/smart-phones/apple-iphone-5\
                "description": "see 'media'"
            },
            "product_name": "Apple iPhone 5",
            "product_code": "12016",
            "_unit_price": "239.00",
            "_line_total": "239.00",
            "extra": {
                "storage": 16,
                "rows": [],
                "product_code": "12016"
            },
            "quantity": 1,
            "order": 5,
            "product": 50
        },
        {
            "line_total": "€ 8.49",
```

Search Result Views

As with the Order View, also the Search Results View is accessible through a CMS page. Say, a search query directed us to http://localhost:8000/en/search/?q=iphone , then the content of this query can be made visible by adding `&format=api` to this URL and get the results in its dictionary structure. This is specially useful to test if a customized search serializer returns the expected results. In our myshop example this may look like:



A generic view to be used for rendering the result list while searching.

```
GET /en/search/?q=iphone&format=api
```

```
HTTP 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "price": "€ 239.00",
            "media": "<div class=\"media-left\"><a href=\"/en/shop/smart-phones/apple-iphone-5\"><
            "product_url": "/en/shop/smart-phones/apple-iphone-5",
            "product_name": "Apple iPhone 5"
        }
    ]
}
```

4.15.2 Final Note

In previous versions of **djangoSHOP**, these kinds of controller implementations had to be implemented by customized Django View classes. This programming pattern led to bloated code, because the programmer had to do a case distinction, whether the request was of type GET, POST or some kind of Ajax. Now **djangoSHOP** is shipped with reusable View classes, and the merchant's implementation must focus exclusively on serializers. This is much easier, because it separates the business logic from the underlying request-response-cycle.

CHAPTER 5

How To's

Some recipes on how to perform certain tasks in **djangoSHOP**.

This collection of recipes unfortunately is not finished yet.

5.1 Add Customized HTML Snippets

When working in *Structure Mode* as provided by **djangoCMS**, while editing the DOM tree inside a placeholder, we might want to add a HTML snippet which is not part of the **Cascade** ecosystem. Instead of creating an additional Django template, it often is much easier to just add a customized plugin. This plugin then is available when editing a placeholder in *Structure Mode*.

5.1.1 Customized Cascade plugin

Creating a customized plugin for the merchant's implementaion of that e-commerce project is very easy. Just add this small Python module:

Listing 5.1: myshop/cascade.py

```python
from cms.plugin_pool import plugin_pool
from shop.cascade.plugin_base import ShopPluginBase

class MySnippetPlugin(ShopPluginBase):
    name = "My Snippet"
    render_template = 'myshop/cascade/my-snippet.html'

plugin_pool.register_plugin(MySnippetPlugin)
```

then, in the project's `settings.py` register that plugin together with all other **Cascade** plugins:

```python
CMSPLUGIN_CASCADE_PLUGINS = (
    'cmsplugin_cascade.segmentation',
    'cmsplugin_cascade.generic',
    'cmsplugin_cascade.link',
    'shop.cascade',
    'cmsplugin_cascade.bootstrap3',
    'myshop.cascade',
    ...
)
```

The template itself `myshop/cascade/my-snippet.html` can contain all templatetags as configured within the Django project.

Often we want to associate customized styles and/or scripts to work with our new template. Since we honor the principle of encapsulation, we somehow must refer to these files in a generic way. This is where django-sekizai helps us:

Listing 5.2: myshop/cascade/my-snippet.html

```
{% load static sekizai_tags %}

{% addtoblock "css" %}<link href="{% static 'myshop/css/my-snippet.css' %}" rel="stylesheet" type="te
{% addtoblock "js" %}<script src="{% static 'myshop/js/my-snippet.js' %}" type="text/javascript"></sc

<div>
    my snippet code goes here...
</div>
```

Note: The main rendering template requires a block such as `{% render_block "css" %}` and `{% render_block "js" %}` which then displays the stylesheets and scripts inside the appropriate HTML elements.

Further customizing the plugin

Sometimes we require additional parameters which shall be customizable by the merchant, while editing the plugin. For **Cascade** this can be achieved very easily. First think about what kind of data to store, and which form widgets are appropriate for that kind of editor. Say we want to add a text field holding the snippets title, then change the change the plugin code from above to:

```
class MySnippetPlugin(ShopPluginBase):
    ...
    glossary_fields = (
        PartialFormField('title',
            widgets.TextInput(),
            label=_("Title")
        ),
    )
```

Inside the rendering template for that plugin, the newly added title can be accessed as:

```
<h1>{{ instance.glossary.title }}</h1>
<div>...
```

Cascade offers many more options than just these. For details please check its reference guide.

5.1.2 Creating a customized Form snippet

Sometimes we might need a dialog form, to store arbitrary information queried from the customer using a customized form. Say we need to know, when to deliver the goods. This information will be stored inside the dictionary `Cart.extra` and thus transferred automatically to `Order.extra` whenever the cart object is converted into an order object.

Our form plugin now must inherit from `shop.cascade.plugin_base.DialogFormPluginBase` instead of our ordinary shop plugin class:

```
from cms.plugin_pool import plugin_pool
from shop.models.cart import CartModel
from shop.cascade.plugin_base import DialogFormPluginBase

class DeliveryDatePlugin(DialogFormPluginBase):
    name = "Delivery Date"
    form_class = 'myshop.forms.DeliveryDateForm'
    render_template = 'myshop/checkout/delivery-date.html'

    def get_form_data(self, context, instance, placeholder):
        cart = CartModel.objects.get_from_request(context['request'])
        initial = {'delivery_date': getattr(cart, 'extra', {}).get('delivery_date', '')}
        return {'initial': initial}

DialogFormPluginBase.register_plugin(DeliveryDatePlugin)
```

here additionally we have to specify a `form_class`. This form class can inherit from `shop.forms.base.DialogForm` or `shop.forms.base.DialogModelForm`. Its behavior is almost identical to its Django's counterparts:

Listing 5.3: myshop/forms.py

```
class DeliveryDateForm(DialogForm):
    scope_prefix = 'data.delivery_date'

    date = fields.DateField(label="Delivery date")

    @classmethod
    def form_factory(cls, request, data, cart):
        delivery_date_form = cls(data=data)
        if delivery_date_form.is_valid():
            cart.extra.update(delivery_date_form.cleaned_data)
        return delivery_date_form
```

The `scope_prefix` marks the JavaScript object below our AngularJS `$scope`. This must be an identifier which is unique across all dialog forms building up our ecosystem of **Cascade** plugins.

The classmethod `form_factory` must, as its name implies, create a form object of the class it belongs to. As in our example from above, we use this to update the cart's `extra` dictionary, whenever the customer submitted a valid delivery date.

The last piece is to put everything together using a form template such as:

Listing 5.4: templates/myshop/checkout/delivery-date.html

```
{% extends "shop/checkout/dialog-base.html" %}

{% block dialog_form %}
<form name="{{ delivery_date_form.form_name }}" novalidate>
    {{ delivery_date_form.as_div }}
</form>
{% endblock %}
```

5.2 Handling Discounts

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

5.3 Taxes

As a general rule, the unit price of a product, shall always contain the net price. When our products show up in the catalog, their method `get_price(request)` is consulted by the framework. Its here where you add tax, depending on the tax model to apply. See below.

5.3.1 Use Cart Modifiers to handle tax

American tax model

European tax model

5.3.2 Other considerations

Try to not reinvent the wheel: Other shop systems / frameworks will contain solutions to this problem. But also ERP-Systems will contain solutions to this problem.

Maybe it is wise to have a look at projects like Tryton (http://tryton.org).

CHAPTER 6

Development and Community

6.1 Changelog for djangoSHOP

6.1.1 0.9.1

- Support for Python 3
- Support for Django-1.9
- Added abstract classes class:*shop.models.delivery.BaseDelivery* and class:*shop.models.delivery.BaseDeliveryItem* for optional partial shipping.

6.1.2 0.9.0

- Separated class:*shop.views.catalog.ProductListView* into its base and the new class class:*shop.views.catalog.CMSPageProductListView* which already has added it appropriate filters.
- Moved `wsgi.py` into upper folder.
- Prototype of `shop.cascade.DialogFormPluginBase.get_form_data` changed. It now accepts `context`, `instance` and `placeholder`.
- Fixed: It was impossible to enter the credit card information for Stripe and then proceed to the next step. Using Stripe was possible only on the last step. This restriction has gone.
- It now also is possible to display a summary of your order before proceeding to the final purchasing step.
- To be more Pythonic, class:*shop.models.cart.CartModelManager* raises a `DoesNotExist` exception instead of `None` for visiting customers.
- Added method `filter_from_request` to class:*shop.models.order.OrderManager*.
- Fixed: OrderAdmin doesn't ignores error if customer URL can't be resolved.
- Fixed: Version checking of Django.
- Fixed: Fieldsets duplication in Product Admin.
- CartPlugin now can be child of ProcessStepPlugin and BootstrapPanelPlugin.
- Added ShopAddToCartPlugin.
- All Checkout Forms now can be rendered as editable or summary.
- All Dialog Forms now can declare a legend.

- In `DialogFormPlugin`, method `form_factory` always returns a form class instead of an error dict if form was invalid.
- Added method `OrderManager.filter_from_request`, which behaves analogous to `CartManager.get_from_request`.
- Fixed lookups using MoneyField by adding method get_prep_value.
- Dropped support for South migrations.
- Fixed: In `ProductIndex`, translations now are always overridden.
- Added class `SyncCatalogView` which can be used to synchronize the cart with a catalog list view.
- Content of Checkout Forms is handled by a single transaction.
- All models such as Product, Order, OrderItem, Cart, CartItem can be overridden by the merchant's implementation. However, we are using the deferred pattern, instead of configuration settings.
- Categories must be implemented as separate **djangoSHOP** addons. However for many implementations pages form the **djangoCMS** can be used as catalog list views.
- The principle on how cart modifiers work, didn't change. There more inversion of control now, in that sense, that now the modifiers decide themselves, how to change the subtotal and final total.
- Existing Payment Providers can be integrated without much hassle.

6.1.3 Since version 0.2.1 a lot of things have changed. Here is a short summary:

- The API of **djangoSHOP** is accessible through a REST interface. This allows us to build MVC on top of that.
- Changed the two OneToOne relations from model Address to User, one was used for shipping, one for billing. Now abstract BaseAddress refers to the User by a single ForeignKey giving the ability to link more than one address to each user. Additionally each address has a priority field for shipping and invoices, so that the latest used address is offered to the client.
- Replaced model shop.models.User by the configuration directive `settings.AUTH_USER_MODEL`, to be compliant with Django documentation.
- The cart now is always stored inside the database; there is no more distinction between session based carts and database carts. Carts for anonymous users are retrieved using the visitor's session_key. Therefore we don't need a utility function such `get_or_create_cart` anymore. Everything is handled by the a new CartManager, which retrieves or creates or cart based on the request session.
- If the quantity of a cart item drops to zero, this items is not automatically removed from the cart. There are plenty of reasons, why it can make sense to have a quantity of zero.
- A WatchList (some say wish-list) has been added. This simply reuses the existing Cart model, where the item quantity is zero.
- Currency and CurrencyField are replaced by Money and MoneyField. These types not only store the amount, but also in which currency this amount is. This has many advantages:
  - An amount is rendered with its currency symbol as a string. This also applies for JSON data-structures, rendered by the REST framework.
  - Money types of different currencies can not be added/substracted by accident. Normal installations woun't be affected, since each shop system must specify its default currency.
- Backend pools for Payment and Shipping have been removed. Instead, a Cart Modifier can inherit from `PaymentModifier` or `ShippingModifier`. This allows to reuse the Cart Modifier Pool for these backends and use the modifiers logic for adding extra rows to he carts total.

- The models `OrderExtraRow` and `OrderItemExtraRow` has been replaced by a JSONField extra_rows in model `OrderModel` and `OrderItemModel`. `OrderAnnotation` now also is stored inside this extra field.
- Renamed for convention with other Django application:
  - date_created -> created_at
  - last_updated -> updated_at
  - ExtraOrderPriceField -> BaseOrderExtraRow
  - ExtraOrderItemPriceField -> BaseItemExtraRow

6.1.4 Version 0.2.1

This is the last release on the old code base. It has been tagged as 0.2.1 and can be examined for historical reasons. Bugs will not be fixed in this release.

6.1.5 Version 0.2.0

- models.FloatField are now automatically localized.
- Support for Django 1.2 and Django 1.3 dropped.
- Product model now has property `can_be_added_to_cart` which is checked before adding the product to cart
- In cart_modifiers methods `get_extra_cart_price_field` and `get_extra_cart_item_price_field` accepts the additional object `request` which can be used to calculate the price according to the state of a session, the IP-address or whatever might be useful. Note for backwards compatibility: Until version 0.1.2, instead of the `request` object, an empty Python dictionary named `state` was passed into the cart modifiers. This `state` object could contain arbitrary data to exchange information between the cart modifiers. This Python dict now is a temporary attribute of the `request` object named `cart_modifier_state`. Use it instead of the `state` object.
- Cart modifiers can add an optional `data` field beside `label` and `value` for both, the ExtraOrderPriceField and the ExtraOrderItemPriceField model. This extra `data` field can contain anything serializable as JSON.

6.1.6 Version 0.1.2

- cart_required and order_required decorators now accept a reversible url name instead and redirect to cart by default
- Added setting *SHOP_PRICE_FORMAT* used in the *priceformat* filter
- Separation of Concern in OrderManager.create_from_cart: It now is easier to extend the Order class with customized data.
- Added OrderConfirmView after the shipping backend views that can be easily extended to display a confirmation page
- Added example payment backend to the example shop
- Added example OrderConfirmView to the example shop
- Unconfirmed orders are now deleted from the database automatically
- **Refactored order status (requires data migration)**

- removed PAYMENT and added CONFIRMING status
- assignment of statuses is now linear
- moved cart.empty() to the PaymentAPI
- orders now store the pk of the originating cart

- **Checkout process works like this:**
  1. CartDetails
  2. **CheckoutSelectionView**
     - POST –> Order.objects.create_from_cart(cart) removes all orders originating from this cart that have status < CONFIRMED(30)
     - creates a new Order with status PROCESSING(10)
  3. **ShippingBackend**
     - self.finished() sets the status to CONFIRMING(20)
  4. **OrderConfirmView**
     - self.confirm_order() sets the status to CONFIRMED(30)
  5. **PaymentBackend**
     - self.confirm_payment() sets the status to COMPLETED(40)
     - empties the related cart
  6. **ThankYouView**
     - does nothing!

6.1.7 Version 0.1.1

- Changed CurrencyField default decimal precision back to 2

6.1.8 Version 0.1.0

- Bumped the CurrencyField precision limitation to 30 max_digits and 10 decimal places, like it should have been since the beginning.
- Made Backends internationalizable, as well as the BillingShippingForm thanks to the introduciton of a new optional backend_verbose_name attribute to backends.
- Added order_required decorator to fix bug #84, which should be used on all payment and shipping views
- Added cart_required decorator that checks for a cart on the checkout view #172
- Added get_product_reference method to Product (for extensibility)
- Cart object is not saved to database if it is empty (#147)
- Before adding items to cart you now have to use get_or_create_cart with save=True
- Changed spelling mistakes in methods from *payed* to *paid* on the Order model and on the API. This is potentially not backwards compatible in some border cases.
- Added a mixin class which helps to localize model fields of type DecimalField in Django admin view.

- Added this newly created mixin class to OrderAdmin, so that all price fields are handled with the correct localization.
- Order status is now directly modified in the shop API
- CartItem URLs were too greedy, they now match less.
- In case a user has two carts, one bound to the session and one to the user, the one from the session will be used (#169)
- Fixed circular import errors by moving base models to shop.models_bases and base managers to shop.models_bases.managers

6.1.9 Version 0.0.13

(Version cleanup)

6.1.10 Version 0.0.12

- Updated translations
- Split urls.py into several sub-files for better readability, and put in a urls shubfolder.
- Made templates extend a common base template
- Using a dynamically generated form for the cart now to validate user input. This will break your cart.html template. Please refer to the changes in cart.html shipped by the shop to see how you can update your own template. Basically you need to iterate over a formset now instead of cart_items.
- Fixed a circular import problem when user overrode their own models

6.1.11 Version 0.0.11

- Performance improvement (update CartItems are now cached to avoid unnecessary db queries)
- Various bugfixes

6.1.12 Version 0.0.10

- New hooks were added to cart modifiers: pre_process_cart and post_process_cart.
- [API change] Cart modifiers cart item methods now recieve a state object, that allows them to pass information between cart modifiers cheaply.
- The cart items are not automatically saved after process_cart_item anymore. This allows for cart modifiers that change the cart's content (also deleting).
- Changed the version definition mechanism. You can now: import shop; shop.__version__. Also, it now conforms to PEP 386
- [API Change] Changed the payment backend API to let get_finished_url and get_cancel_url return strings instead of HttpResponse objects (this was confusing)
- Tests for the shop are now runnable from any project
- added URL to CartItemView.delete()

6.1.13 Version 0.0.9

- Changed the base class for Cart Modifiers. Methods are now expected to return a tuple, and not direectly append it to the extra_price_fields. Computation of the total is not done using an intermediate "current_total" attribute.
- Added a SHOP_FORCE_LOGIN setting that restricts the checkout process to loged-in users.

6.1.14 Version 0.0.8

- Major change in the way injecting models for extensibility works: the base models are now abstract, and the shop provides a set of default implementations that users can replace / override using the settings, as usual. A special mechanism is required to make the Foreign keys to shop models work. This is explained in shop.utils.loaders

6.1.15 Version 0.0.7

- Fixed bug in the extensibility section of CartItem
- Added complete German translations
- Added verbose names to the Address model in order to have shipping and billing forms that has multilingual labels.

6.1.16 Version 0.0.6

(Bugfix release)

- Various bugfixes
- Creating AddressModels for use with the checkout view (the default ones at least) were bugged, and would spawn new instances on form post, instead of updating the user's already existing ones.
- Removed redundant payment method field on the Order model.
- The "thank you" view does not crash anymore when it's refreshed. It now displays the last order the user placed.
- Fixed a bug in the shippingbilling view where the returned form was a from class instead of a from instance.

6.1.17 Version 0.0.5

- Fix a bug in 0.0.4 that made South migration fail with Django < 1.3

6.1.18 Version 0.0.4

- Addresses are now stored as one single text field on the Order objects
- OrderItems now have a ForeignKey relation to Products (to retrieve the product more easily)
- New templatetag ("products")
- Made most models swappable using settings (see docs)
- Changed checkout views. The shop uses one single checkout view by default now.
- Created new mechanism to use custom Address models (see docs)
- Moved all Address-related models to shop.addressmodel sub-app

- Removed Client Class
- Removed Product.long_description and Product.short_description from the Product superclass
- Bugfixes, docs update

6.1.19 Version 0.0.3

- More packaging fixes (missing templates, basically)

6.1.20 Version 0.0.2

- Packaging fix (added MANIFEST.in)

6.1.21 Version 0.0.1

- Initial release to Pypi

6.2 Contributing

6.2.1 Naming conventions

The official name of this project is **djangoSHOP**. Third party plugins for **djangoSHOP** shall follow the same naming convention as for plugins of **djangoCMS**: Third party package names shall start with **djangoshop** followed by a dash; no space shall be added between **django** and **shop**, for example: `djangoshop-stripe`

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

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

6.2.2 Running tests

It's important to run tests before committing :)

Setting up the environment

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

1. Clone the repository and cd into it:

```
git clone https://github.com/awesto/django-shop
cd django-shop
```

2. Create a virtualenv, and activate it:

```
virtualenv ~/.virtualenvs/django-shop
source ~/.virtualenvs/django-shop/bin/activate
```

3. Install the project in development mode:

```
pip install -e .
```

4. Install the development requirements:

```
pip install -r requirements/django18/testing.txt
```

That's it! Now, you should be able to run the tests:

```
py.test tests
```

We use tox as a CI tool. To run the full CI test suite and get a coverage report, all you have to do is this:

```
pip install tox
tox
```

If you work on a certain part of the code base and you want to run the related tests, you may only want to run the tests affecting that part. In such a case use `py.test` from your testing environment and specify the file to test, or for more granularity the class name or even the method name. Here are two examples:

```
py.test testshop/test_money.py
py.test testshop/test_money.py -k test_pickle
```

Measuring which lines of code have been "seen" be the test runner is an important task while testing. Do this by creating a coverage report, for example with:

```
coverage run $(which py.test) testshop
coverage report
```

or if you to test only a specific class

coverage run .tox/py27-django19/bin/py.test testshop/test_money.py coverage report -m shop/money/*.py

Note: Using tox and py.test is optional. If you prefer the conventional way of running tests, you can do this: `django-admin.py test tests --settings shop.testsettings`

6.2.3 Community

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

We also have a mailing list and a google group:

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

6.2.4 Code guidelines

Unless otherwise specified, follow **PEP 8** as closely as possible.

An exception to PEP 8 is our rules on line lengths. Don't limit lines of code to 79 characters if it means the code looks significantly uglier or is harder to read. Consider 100 characters as a soft, and 119 as a hard limit. Here soft limit means, that unless a line must be splitted across two lines, it is more readable to stay with a long line.

Use the issue tracker only to report bugs. Send unsolicited pull requests only to fix bug – never to add new features.

Use stack-overflow to ask for questions related to **djangoSHOP**.

Most pull requests will be rejected without proper unit testing.

Before adding a new feature, please write a specification using the style for Django Enhancement Proposals.

More information about how to send a Pull Request can be found on GitHub: http://help.github.com/send-pull-requests/

CHAPTER 7

License

DjangoSHOP is licensed under the terms of the BSD license.