Building a basic shopping cart application using Django framework

Creating a Shopping Cart project

We are going to start with a new Django project to build an online shop. Our users will be able to browse through a product catalog and add products to a shopping cart. Finally, they will be able to check out the cart and place an order. This chapter will cover the following functionalities of an online shop:

- Creating the product catalog models, adding them to the administration site, and building the basic views to display the catalog
- Building a shopping cart system using Django sessions to allow users to keep selected products while they browse the site
- Creating the form and functionality to place orders on the site
- Sending an asynchronous email confirmation to users when they place an order

Open a shell, create a virtual environment for the new project, and activate it with the following commands:

```
mkdir env
virtualenv env/myshop
source env/myshop/bin/activate
```

Install Django in your virtual environment with the following command:

```
pip install Django==2.0.5
```

Start a new project called `myshop` with an application called `shop` by opening a shell and running the following commands:

```
django-admin startproject myshop
cd myshop/
django-admin startapp shop
```

Edit the `settings.py` file of your project and add the `shop` application to the `INSTALLED_APPS` setting as follows:

```
INSTALLED_APPS = [
    # ...
    'shop.apps.ShopConfig',
]
```

Your application is now active for this project. Let's define the models for the product catalog.

Creating product catalog models

The catalog of our shop will consist of products that are organized into different categories. Each product will have a name, optional description, optional image, price, and availability. Edit the `models.py` file of the `shop` application that you just created and add the following code:

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200,
                            db_index=True)
    slug = models.SlugField(max_length=200,
                            unique=True)

    class Meta:
        ordering = ('name',)
        verbose_name = 'category'
        verbose_name_plural = 'categories'

    def __str__(self):
        return self.name

class Product(models.Model):
    category = models.ForeignKey(Category,
                                 related_name='products',
                                 on_delete=models.CASCADE)
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True)
    image = models.ImageField(upload_to='products/%Y/%m/%d',
                              blank=True)
    description = models.TextField(blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    class Meta:
        ordering = ('name',)
        index_together = (('id', 'slug'),)

    def __str__(self):
        return self.name
```

These are the `Category` and `Product` models. The `Category` model consists of a `name` field and a `slug` unique field (`unique` implies the creation of an index). The `Product` model fields are as follows:

- `category`: `ForeignKey` to the `Category` model. This is a many-to-one relationship: a product belongs to one category and a category contains multiple products.
- `name`: The name of the product.
- `slug`: The slug for this product to build beautiful URLs.
- `image`: An optional product image.
- `description`: An optional description of the product.
- `price`: This field uses Python's `decimal.Decimal` type to store a fixed-precision decimal number. The maximum number of digits (including the decimal places) is set using the `max_digits` attribute and decimal places with the `decimal_places` attribute.
- `available`: A boolean value that indicates whether the product is available or not. It will be used to enable/disable the product in the catalog.
- `created`: This field stores when the object was created.
- `updated`: This field stores when the object was last updated.

For the `price` field, we use `DecimalField` instead of `FloatField` to avoid rounding issues.

> *Always use `DecimalField` to store monetary amounts. `FloatField` uses Python's `float` type internally, whereas `DecimalField` uses Python's `Decimal` type. By using the `Decimal` type, you will avoid `float` rounding issues.*

In the `Meta` class of the `Product` model, we use the `index_together` meta option to specify an index for the `id` and `slug` fields together. We define this index because we plan to query products by both `id` and `slug`. Both fields are indexed together to improve performances for queries that utilize the two fields.

Since we are going to deal with images in our models, open the shell and install `Pillow` with the following command:

```
pip install Pillow==5.1.0
```

Now, run the next command to create initial migrations for your project:

```
python manage.py makemigrations
```

You will see the following output:

```
Migrations for 'shop':
  shop/migrations/0001_initial.py
    - Create model Category
    - Create model Product
    - Alter index_together for product (1 constraint(s))
```

Run the next command to sync the database:

```
python manage.py migrate
```

You will see output that includes the following line:

```
Applying shop.0001_initial... OK
```

The database is now synced with your models.

Registering catalog models on the admin site

Let's add our models to the administration site so that we can easily manage categories and products. Edit the `admin.py` file of the `shop` application and add the following code to it:

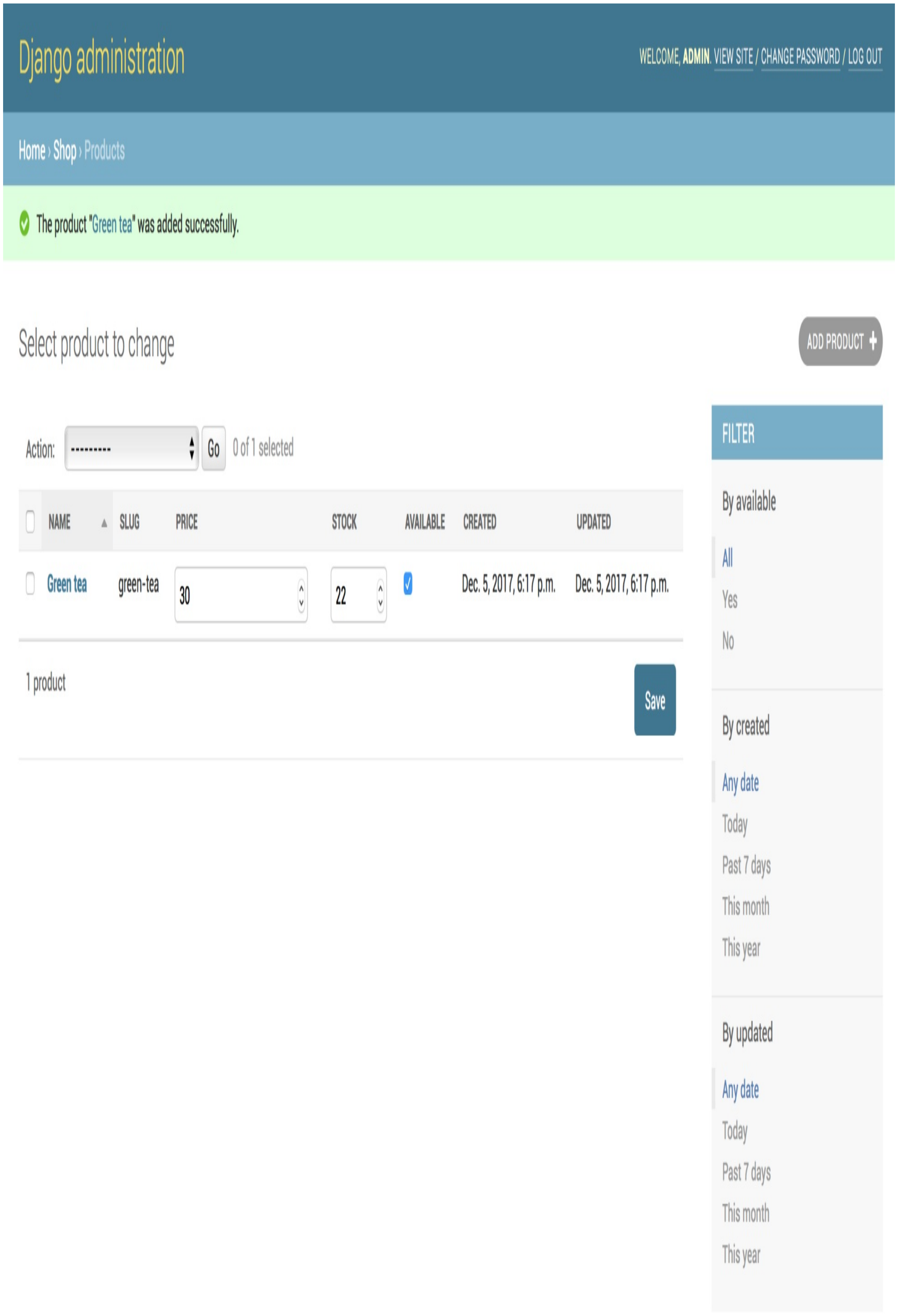
```
from django.contrib import admin
from .models import Category, Product

@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug']
    prepopulated_fields = {'slug': ('name',)}

@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price',
                    'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'available']
    prepopulated_fields = {'slug': ('name',)}
```

Remember that we use the `prepopulated_fields` attribute to specify fields where the value is automatically set using the value of other fields. As you have seen before, this is convenient for generating slugs. We use the `list_editable` attribute in the `ProductAdmin` class to set the fields that can be edited from the list display page of the administration site. This will allow you to edit multiple rows at once. Any field in `list_editable` must also be listed in the `list_display` attribute since only the fields displayed can be edited.

Now, create a superuser for your site using the following command:

```
python manage.py createsuperuser
```

Start the development server with the command `python manage.py runserver`. Open `http://127.0.0.1:8000/admin/shop/product/add/` in your browser and log in with the user that you just created. Add a new category and product using the administration interface. The product change list page of the administration page will then look like this:



Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Shop › Products

The product "Green tea" was added successfully.

Select product to change

ADD PRODUCT

Action: --------- Go 0 of 1 selected

	NAME	SLUG	PRICE	STOCK	AVAILABLE	CREATED	UPDATED
☐	Green tea	green-tea	30	22	✓	Dec. 5, 2017, 6:17 p.m.	Dec. 5, 2017, 6:17 p.m.

1 product

Save

FILTER

By available
- All
- Yes
- No

By created
- Any date
- Today
- Past 7 days
- This month
- This year

By updated
- Any date
- Today
- Past 7 days
- This month
- This year

Building catalog views

In order to display the product catalog, we need to create a view to list all the products or filter products by a given category. Edit the `views.py` file of the `shop` application and add the following code to it:

```
from django.shortcuts import render, get_object_or_404
from .models import Category, Product

def product_list(request, category_slug=None):
    category = None
    categories = Category.objects.all()
    products = Product.objects.filter(available=True)
    if category_slug:
        category = get_object_or_404(Category, slug=category_slug)
        products = products.filter(category=category)
    return render(request,
                  'shop/product/list.html',
                  {'category': category,
                   'categories': categories,
                   'products': products})
```

We will filter the QuerySet with `available=True` to retrieve only available products. We use an optional `category_slug` parameter to optionally filter products by a given category.

We also need a view to retrieve and display a single product. Add the following view to the `views.py` file:

```
def product_detail(request, id, slug):
    product = get_object_or_404(Product,
                                id=id,
                                slug=slug,
                                available=True)
    return render(request,
                  'shop/product/detail.html',
                  {'product': product})
```

The `product_detail` view expects the `id` and `slug` parameters in order to retrieve the `Product` instance. We can get this instance just through the ID since it's a unique attribute. However, we include the slug in the URL to build SEO-friendly URLs for products.

After building the product list and detail views, we have to define URL patterns for them. Create a new file inside the `shop` application directory and name it `urls.py`. Add the following code to it:

```
from django.urls import path
from . import views

app_name = 'shop'

urlpatterns = [
    path('', views.product_list, name='product_list'),
    path('<slug:category_slug>/', views.product_list,
         name='product_list_by_category'),
    path('<int:id>/<slug:slug>/', views.product_detail,
         name='product_detail'),
]
```

These are the URL patterns for our product catalog. We have defined two different URL patterns for the `product_list` view: a pattern named `product_list`, which calls the `product_list` view without any parameters; and a pattern named `product_list_by_category`, which provides a `category_slug` parameter to the view for filtering products according to a given category. We added a pattern for the `product_detail` view, which passes the `id` and `slug` parameters to the view in order to retrieve a specific product.

Edit the `urls.py` file of the `myshop` project to make it look like this:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('shop.urls', namespace='shop')),
]
```

In the main URL patterns of the project, we will include URLs for the `shop` application under a custom namespace named `shop`.

Now, edit the `models.py` file of the `shop` application, import the `reverse()` function, and add a `get_absolute_url()` method to the `Category` and `Product` models as follows:

```
from django.urls import reverse
# ...
class Category(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('shop:product_list_by_category',
                       args=[self.slug])

class Product(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('shop:product_detail',
                       args=[self.id, self.slug])
```

As you already know, `get_absolute_url()` is the convention to retrieve the URL for a given object. Here, we will use the URLs patterns that we just defined in the `urls.py` file.

Creating catalog templates

Now, we need to create templates for the product list and detail views. Create the following directory and file structure inside the `shop` application directory:

```
templates/
    shop/
        base.html
        product/
            list.html
            detail.html
```

We need to define a base template, and then extend it in the product list and detail templates. Edit the `shop/base.html` template and add the following code to it:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>{% block title %}My shop{% endblock %}</title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
  <div id="header">
    <a href="/" class="logo">My shop</a>
  </div>
  <div id="subheader">
    <div class="cart">
      Your cart is empty.
    </div>
  </div>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
</body>
```

```
</html>
```

This is the base template that we will use for our shop. In order to include the CSS styles and images that are used by the templates, you will need to copy the static files that accompany this chapter, located in the `static/` directory of the `shop` application. Copy them to the same location in your project.

Edit the `shop/product/list.html` template and add the following code to it:

```
{% extends "shop/base.html" %}
{% load static %}

{% block title %}
  {% if category %}{{ category.name }}{% else %}Products{% endif %}
{% endblock %}

{% block content %}
  <div id="sidebar">
    <h3>Categories</h3>
    <ul>
      <li {% if not category %}class="selected"{% endif %}>
        <a href="{% url "shop:product_list" %}">All</a>
      </li>
      {% for c in categories %}
        <li {% if category.slug == c.slug %}class="selected"
        {% endif %}>
          <a href="{{ c.get_absolute_url }}">{{ c.name }}</a>
        </li>
      {% endfor %}
    </ul>
  </div>
  <div id="main" class="product-list">
    <h1>{% if category %}{{ category.name }}{% else %}Products
    {% endif %}</h1>
    {% for product in products %}
      <div class="item">
        <a href="{{ product.get_absolute_url }}">
          <img src="{% if product.image %}{{ product.image.url }}{%
          else %}{% static "img/no_image.png" %}{% endif %}">
        </a>
        <a href="{{ product.get_absolute_url }}">{{ product.name }}</a>
        <br>
        ${{ product.price }}
      </div>
    {% endfor %}
  </div>
{% endblock %}
```

This is the product list template. It extends the `shop/base.html` template and uses the `categories` context variable to display all the categories in a sidebar and `products` to display the products of the current page. The same template is used for both: listing all available products and listing products filtered by a category. Since the `image` field of the `Product` model can be blank, we need to provide a default image for the products that don't have an image. The image is located in our static files directory with the relative path `img/no_image.png`.

Since we are using `ImageField` to store product images, we need the development server to serve uploaded image files.

Edit the `settings.py` file of `myshop` and add the following settings:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

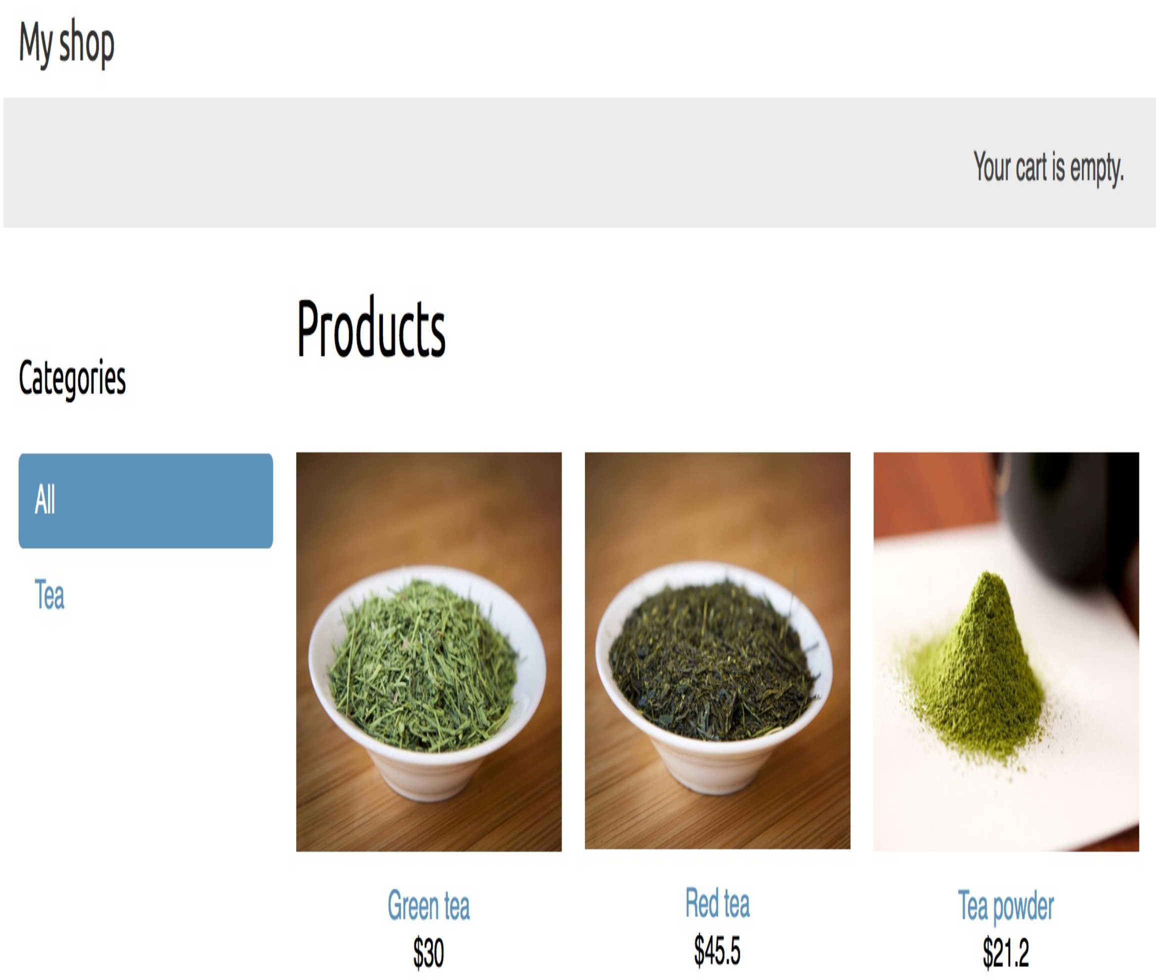
`MEDIA_URL` is the base URL that serves media files uploaded by users. `MEDIA_ROOT` is the local path where these files reside, which we build by dynamically prepending the `BASE_DIR` variable.

For Django to serve the uploaded media files using the development server, edit the main `urls.py` file of `myshop` and add the following code to it:

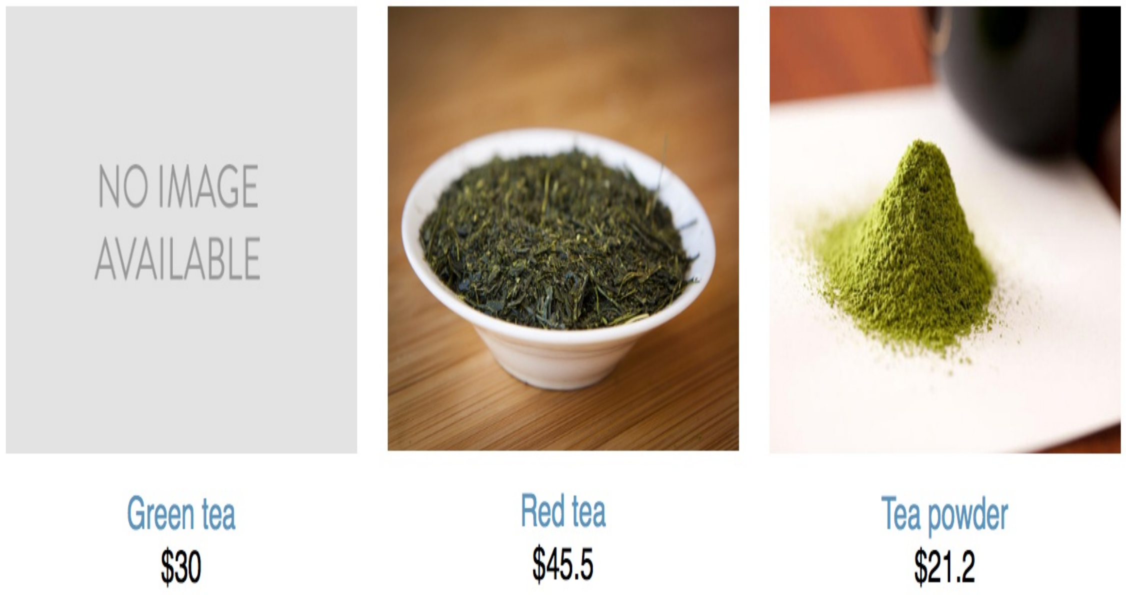
```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ...
]
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

Remember that we only serve static files this way during development. In a production environment, you should never serve static files with Django.

Add a couple of products to your shop using the administration site and open `http://127.0.0.1:8000/` in your browser. You will see the product list page, which looks like this:



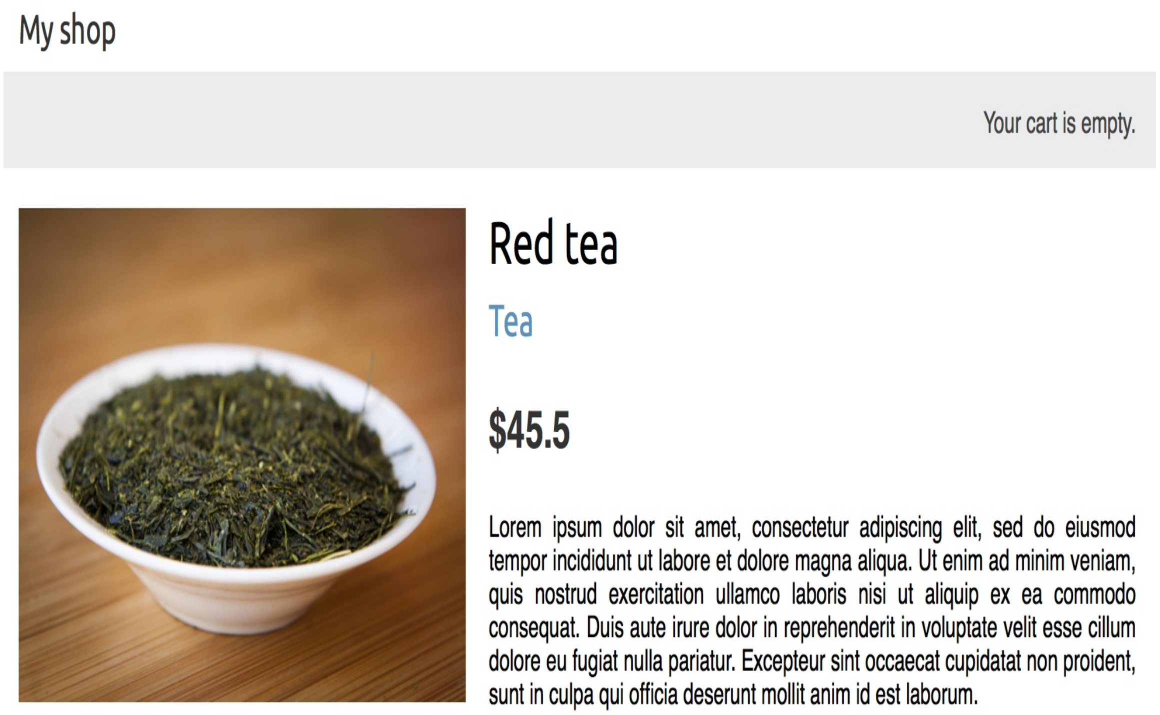
If you create a product using the administration site and don't upload any image for it, the default `no_image.png` image will be displayed instead:



Let's edit the product detail template. Edit the `shop/product/detail.html` template and add the following code to it:

```
{% extends "shop/base.html" %}
{% load static %}

{% block title %}
  {{ product.name }}
{% endblock %}

{% block content %}
  <div class="product-detail">
    <img src="{% if product.image %}{{ product.image.url }}{% else %}
    {% static "img/no_image.png" %}{% endif %}">
    <h1>{{ product.name }}</h1>
    <h2><a href="{{ product.category.get_absolute_url }}">{{
    product.category }}</a></h2>
    <p class="price">${{ product.price }}</p>
    {{ product.description|linebreaks }}
  </div>
{% endblock %}
```

We call the `get_absolute_url()` method on the related category object to display the available products that belong to the same category. Now, open `http://127.0.0.1:8000/` in your browser and click on any product to see the product detail page. It will look as follows:



My shop

Your cart is empty.

Red tea

Tea

$45.5

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

You have now created a basic product catalog.

Building a shopping cart

After building the product catalog, the next step is to create a shopping cart so that users can pick the products that they want to purchase. A shopping cart allows users to select products and set the amounts they want to order, and then store this information temporarily, while they browse the site until they eventually place an order. The cart has to be persisted in the session so that the cart items are maintained during the user's visit.

We will use Django's session framework to persist the cart. The cart will be kept in the session until it finishes or the user checks out of the cart. We will also need to build additional Django models for the cart and its items.

Using Django sessions

Django provides a session framework that supports anonymous and user sessions. The session framework allows you to store arbitrary data for each visitor. Session data is stored on the server side, and cookies contain the session ID unless you use the cookie-based session engine. The session middleware manages the sending and receiving of cookies. The default session engine stores session data in the database, but you can choose between different session engines.

To use sessions, you have to make sure that the `MIDDLEWARE` setting of your project contains `'django.contrib.sessions.middleware.SessionMiddleware'`. This middleware manages sessions. It's added by default to the `MIDDLEWARE` setting when you create a new project using the `startproject` command.

The session middleware makes the current session available in the `request` object. You can access the current session using `request.session`, treating it like a Python dictionary to store and retrieve session data. The session dictionary accepts any Python object by default that can be serialized to JSON. You can set a variable in the session like this:

```
request.session['foo'] = 'bar'
```

Retrieve a session key as follows:

```
request.session.get('foo')
```

Delete a key you previously stored in the session as follows:

```
del request.session['foo']
```

You can just treat `request.session` like a standard Python dictionary.

When users log in to the site, their anonymous session is lost and a new session is created for the authenticated users. If you store items in an anonymous session that you need to keep after the user logs in, you will have to copy the old session data into the new session.

Session settings

There are several settings you can use to configure sessions for your project. The most important is `SESSION_ENGINE`. This setting allows you to set the place where sessions are stored. By default, Django stores sessions in the database using the `Session` model of the `django.contrib.sessions` application.

Django offers the following options for storing session data:

- **Database sessions**: Session data is stored in the database. This is the default session engine.
- **File-based sessions**: Session data is stored in the filesystem.
- **Cached sessions**: Session data is stored in a cache backend. You can specify cache backends using the `CACHES` setting. Storing session data in a cache system provides the best performance.
- **Cached database sessions**: Session data is stored in a write-through cache and database. Reads-only use the database if the data is not already in the cache.
- **Cookie-based sessions**: Session data is stored in the cookies that are sent to the browser.

For better performance, use a cache-based session engine. Django supports Memcached out of the box and you can find third-party cache backends for Redis and other cache systems.

You can customize sessions with specific settings. Here are some of the important session-related settings:

- `SESSION_COOKIE_AGE`: The duration of session cookies in seconds. The default value is `1209600` (two weeks).
- `SESSION_COOKIE_DOMAIN`: The domain used for session cookies. Set this to `mydomain.com` to enable cross-domain cookies or use `None` for a standard domain cookie.
- `SESSION_COOKIE_SECURE`: A boolean indicating that the cookie should only be sent if the connection is an HTTPS connection.
- `SESSION_EXPIRE_AT_BROWSER_CLOSE`: A boolean indicating that the session has to expire when the browser is closed.
- `SESSION_SAVE_EVERY_REQUEST`: A boolean that, if `True`, will save the session to the database on every request. The session expiration is also updated each time it's saved.

You can see all the session settings and their default values at `https://docs.djangoproject.com/en/2.0/ref/settings/#sessions`.

Session expiration

You can choose to use browser-length sessions or persistent sessions using the `SESSION_EXPIRE_AT_BROWSER_CLOSE` setting. This is set to `False` by default, forcing the session duration to the value stored in the `SESSION_COOKIE_AGE` setting. If you set `SESSION_EXPIRE_AT_BROWSER_CLOSE` to `True`, the session will expire when the user closes the browser, and the `SESSION_COOKIE_AGE` setting will not have any effect.

You can use the `set_expiry()` method of `request.session` to overwrite the duration of the current session.

Storing shopping carts in sessions

We need to create a simple structure that can be serialized to JSON for storing cart items in a session. The cart has to include the following data for each item contained in it:

- The ID of a `Product` instance
- Quantity selected for the product
- Unit price for the product

Since product prices may vary, we take the approach of storing the product's price along with the product itself when it's added to the cart. By doing so, we use the current price of the product when users add it to their cart, no matter if the product's price is changed afterwards.

Now, you have to build functionality to create carts and associate them with sessions. The shopping cart has to work as follows:

- When a cart is needed, we check if a custom session key is set. If no cart is set in the session, we create a new cart and save it in the cart session key.
- For successive requests, we perform the same check and get the cart items from the cart session key. We retrieve the cart items from the session and their related `Product` objects from the database.

Edit the settings.py file of your project and add the following setting to it:

```
CART_SESSION_ID = 'cart'
```

This is the key that we are going to use to store the cart in the user session. Since Django sessions are managed per-visitor, we can use the same cart session key for all sessions.

Let's create an application for managing shopping carts. Open the Terminal and create a new application, running the following command from the project directory:

```
python manage.py startapp cart
```

Then, edit the settings.py file of your project and add the new application to the INSTALLED_APPS setting as follows:

```
INSTALLED_APPS = [
    # ...
    'shop.apps.ShopConfig',
    'cart.apps.CartConfig',
]
```

Create a new file inside the cart application directory and name it cart.py. Add the following code to it:

```
from decimal import Decimal
from django.conf import settings
from shop.models import Product

class Cart(object):

    def __init__(self, request):
        """
        Initialize the cart.
        """
        self.session = request.session
        cart = self.session.get(settings.CART_SESSION_ID)
        if not cart:
            # save an empty cart in the session
            cart = self.session[settings.CART_SESSION_ID] = {}
        self.cart = cart
```

This is the `Cart` class that will allow us to manage the shopping cart. We require the cart to be initialized with a `request` object. We store the current session using `self.session = request.session` to make it accessible to the other methods of the `Cart` class. First, we try to get the cart from the current session using `self.session.get(settings.CART_SESSION_ID)`. If no cart is present in the session, we create an empty cart by setting an empty dictionary in the session. We expect our cart dictionary to use product IDs as keys and a dictionary with quantity and price as the value for each key. By doing so, we can guarantee that a product is not added more than once in the cart; this way we also simplify the way to retrieve cart items.

Let's create a method to add products to the cart or update their quantity. Add the following `add()` and `save()` methods to the `Cart` class:

```
class Cart(object):
    # ...
    def add(self, product, quantity=1, update_quantity=False):
        """
        Add a product to the cart or update its quantity.
        """
        product_id = str(product.id)
        if product_id not in self.cart:
            self.cart[product_id] = {'quantity': 0,
                                      'price': str(product.price)}
        if update_quantity:
            self.cart[product_id]['quantity'] = quantity
        else:
            self.cart[product_id]['quantity'] += quantity
        self.save()

    def save(self):
        # mark the session as "modified" to make sure it gets saved
        self.session.modified = True
```

The `add()` method takes the following parameters as input:

- `product`: The `product` instance to add or update in the cart.
- `quantity`: An optional integer with the product quantity. This defaults to `1`.
- `update_quantity`: This is a boolean that indicates whether the quantity needs to be updated with the given quantity (`True`), or whether the new quantity has to be added to the existing quantity (`False`).

We use the product ID as a key in the cart's content dictionary. We convert the product ID into a string because Django uses JSON to serialize session data, and JSON only allows string key names. The product ID is the key and the value that we persist is a dictionary with quantity and price figures for the product. The product's price is converted from decimal into a string in order to serialize it. Finally, we call the `save()` method to save the cart in the session.

The `save()` method marks the session as modified using `session.modified = True`. This tells Django that the session has changed and needs to be saved.

We also need a method for removing products from the cart. Add the following method to the `Cart` class:

```
class Cart(object):
    # ...
    def remove(self, product):
        """
        Remove a product from the cart.
        """
        product_id = str(product.id)
        if product_id in self.cart:
            del self.cart[product_id]
            self.save()
```

The remove() method removes a given product from the cart dictionary and calls the save() method to update the cart in the session.

We will have to iterate through the items contained in the cart and access the related Product instances. To do so, you can define an __iter__() method in your class. Add the following method to the Cart class:

```
class Cart(object):
    # ...
    def __iter__(self):
        """
        Iterate over the items in the cart and get the products
        from the database.
        """
        product_ids = self.cart.keys()
        # get the product objects and add them to the cart
        products = Product.objects.filter(id__in=product_ids)

        cart = self.cart.copy()
        for product in products:
            cart[str(product.id)]['product'] = product

        for item in cart.values():
            item['price'] = Decimal(item['price'])
            item['total_price'] = item['price'] * item['quantity']
            yield item
```

In the __iter__() method, we retrieve the Product instances that are present in the cart to include them in the cart items. We copy the current cart in the cart variable and add the Product instances to it. Finally, we iterate over the cart items, converting the item's price back into decimal, and add a total_price attribute to each item. Now, we can easily iterate over the items in the cart.

We also need a way to return the number of total items in the cart. When the len() function is executed on an object, Python calls its __len__() method to retrieve its length. We are going to define a custom __len__() method to return the total number of items stored in the cart. Add the following __len__() method to the

`Cart` class:

```
class Cart(object):
    # ...
    def __len__(self):
        """
        Count all items in the cart.
        """
        return sum(item['quantity'] for item in self.cart.values())
```

We return the sum of the quantities of all the cart items.

Add the following method to calculate the total cost of the items in the cart:

```
class Cart(object):
    # ...
    def get_total_price(self):
        return sum(Decimal(item['price']) * item['quantity'] for item in
self.cart.values())
```

And finally, add a method to clear the cart session:

```
class Cart(object):
    # ...
    def clear(self):
        # remove cart from session
        del self.session[settings.CART_SESSION_ID]
        self.save()
```

Our `Cart` class is now ready to manage shopping carts.

Creating shopping cart views

Now that we have a `Cart` class to manage the cart, we need to create the views to add, update, or remove items from it. We need to create the following views:

- A view to add or update items in a cart, which can handle current and new quantities
- A view to remove items from the cart
- A view to display cart items and totals

Adding items to the cart

In order to add items to the cart, we need a form that allows the user to select a quantity. Create a `forms.py` file inside the `cart` application directory and add the following code to it:

```
from django import forms

PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]

class CartAddProductForm(forms.Form):
    quantity = forms.TypedChoiceField(
                                choices=PRODUCT_QUANTITY_CHOICES,
                                coerce=int)
    update = forms.BooleanField(required=False,
                                initial=False,
                                widget=forms.HiddenInput)
```

We will use this form to add products to the cart. Our `CartAddProductForm` class contains the following two fields:

- `quantity`: This allows the user to select a quantity between 1-20. We use a `TypedChoiceField` field with `coerce=int` to convert the input into an integer.
- `update`: This allows you to indicate whether the quantity has to be added to any existing quantity in the cart for this product (`False`), or whether the existing quantity has to be updated with the given quantity (`True`). We use a `HiddenInput` widget for this field since we don't want to display it to the user.

Let's create a view for adding items to the cart. Edit the `views.py` file of the `cart` application and add the following code to it:

```
from django.shortcuts import render, redirect, get_object_or_404
from django.views.decorators.http import require_POST
from shop.models import Product
from .cart import Cart
from .forms import CartAddProductForm

@require_POST
def cart_add(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    form = CartAddProductForm(request.POST)
    if form.is_valid():
        cd = form.cleaned_data
        cart.add(product=product,
                 quantity=cd['quantity'],
                 update_quantity=cd['update'])
    return redirect('cart:cart_detail')
```

This is the view for adding products to the cart or updating quantities for existing products. We use the `require_POST` decorator to allow only `POST` requests, since this view is going to change data. The view receives the product ID as a parameter. We retrieve the `Product` instance with the given ID and validate `CartAddProductForm`. If the form is valid, we either add or update the product in the cart. The view redirects to the `cart_detail` URL that will display the content of the cart. We are going to create the `cart_detail` view shortly.

We also need a view to remove items from the cart. Add the following code to the `views.py` file of the `cart` application:

```
def cart_remove(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    cart.remove(product)
    return redirect('cart:cart_detail')
```

The `cart_remove` view receives the product ID as a parameter. We retrieve the `Product` instance with the given ID and remove the product from the cart. Then, we redirect the user to the `cart_detail` URL.

Finally, we need a view to display the cart and its items. Add the following view to the `views.py` file of the `cart` application:

```
def cart_detail(request):
    cart = Cart(request)
    return render(request, 'cart/detail.html', {'cart': cart})
```

The `cart_detail` view gets the current cart to display it.

We have created views to add items to the cart, update quantities, remove items from the cart, and display the cart content. Let's add URL patterns for these views. Create a new file inside the `cart` application directory and name it `urls.py`. Add the following URLs to it:

```
from django.urls import path
from . import views

app_name = 'cart'

urlpatterns = [
    path('', views.cart_detail, name='cart_detail'),
    path('add/<int:product_id>/',
         views.cart_add,
         name='cart_add'),
    path('remove/<int:product_id>/',
         views.cart_remove,
         name='cart_remove'),
]
```

Edit the main `urls.py` file of the `myshop` project and add the following URL pattern to include the cart URLs:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('cart/', include('cart.urls', namespace='cart')),
    path('', include('shop.urls', namespace='shop')),
]
```

Make sure that you include this URL pattern before the `shop.urls` pattern, since it's more restrictive than the latter.

Building a template to display the cart

The `cart_add` and `cart_remove` views don't render any templates, but we need to create a template for the `cart_detail` view to display cart items and totals.

Create the following file structure inside the `cart` application directory:

```
templates/
    cart/
        detail.html
```

Edit the `cart/detail.html` template and add the following code to it:

```
{% extends "shop/base.html" %}
{% load static %}

{% block title %}
  Your shopping cart
{% endblock %}

{% block content %}
  <h1>Your shopping cart</h1>
  <table class="cart">
    <thead>
      <tr>
        <th>Image</th>
        <th>Product</th>
        <th>Quantity</th>
        <th>Remove</th>
        <th>Unit price</th>
        <th>Price</th>
      </tr>
    </thead>
    <tbody>
```

```
      {% for item in cart %}
        {% with product=item.product %}
          <tr>
            <td>
              <a href="{{ product.get_absolute_url }}">
                <img src="{% if product.image %}{{ product.image.url }}
                {% else %}{% static "img/no_image.png" %}{% endif %}">
              </a>
            </td>
            <td>{{ product.name }}</td>
            <td>{{ item.quantity }}</td>
            <td><a href="{% url "cart:cart_remove" product.id
            %}">Remove</a></td>
            <td class="num">${{ item.price }}</td>
            <td class="num">${{ item.total_price }}</td>
          </tr>
        {% endwith %}
      {% endfor %}
      <tr class="total">
        <td>Total</td>
        <td colspan="4"></td>
        <td class="num">${{ cart.get_total_price }}</td>
      </tr>
    </tbody>
  </table>
  <p class="text-right">
    <a href="{% url "shop:product_list" %}" class="button
    light">Continue shopping</a>
    <a href="#" class="button">Checkout</a>
  </p>
{% endblock %}
```

This is the template that is used to display the cart content. It contains a table with the items stored in the current cart. We allow users to change the quantity of the selected products using a form that is posted to the `cart_add` view. We also allow users to remove items from the cart by providing a Remove link for each of them.

Adding products to the cart

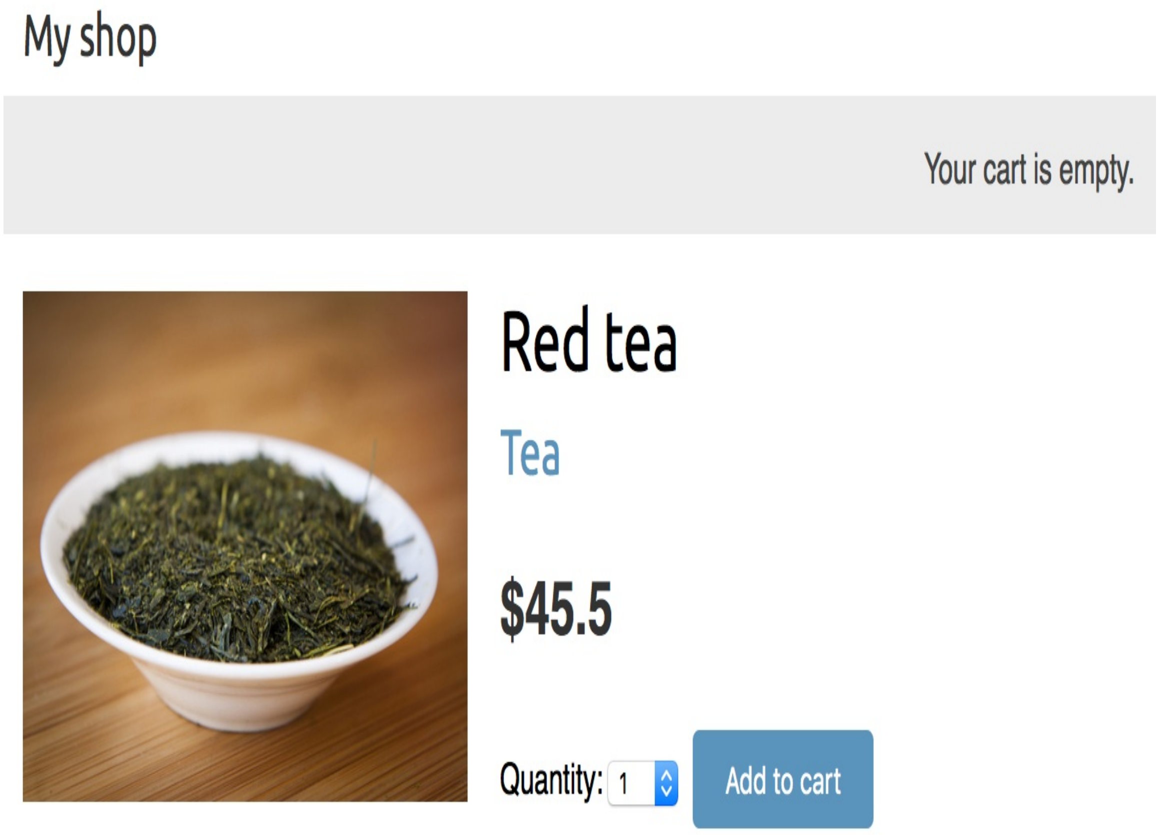
Now, we need to add an Add to cart button to the product detail page. Edit the `views.py` file of the `shop` application, and add `CartAddProductForm` to the `product_detail` view as follows:

```
from cart.forms import CartAddProductForm

def product_detail(request, id, slug):
    product = get_object_or_404(Product, id=id,
                                         slug=slug,
                                         available=True)
    cart_product_form = CartAddProductForm()
    return render(request,
                  'shop/product/detail.html',
                  {'product': product,
                   'cart_product_form': cart_product_form})
```

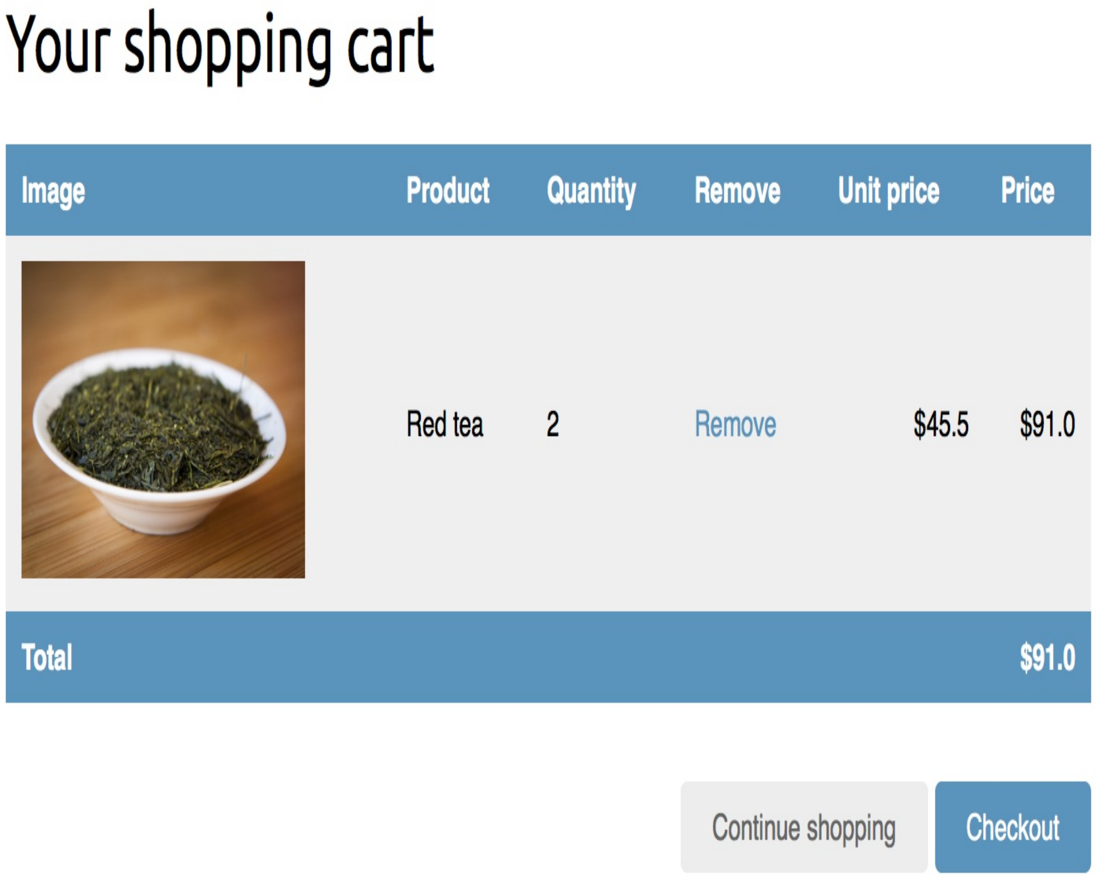
Edit the `shop/product/detail.html` template of the `shop` application, and add the following form to the product's price as follows:

```
<p class="price">${{ product.price }}</p>
<form action="{% url "cart:cart_add" product.id %}" method="post">
  {{ cart_product_form }}
  {% csrf_token %}
  <input type="submit" value="Add to cart">
</form>
{{ product.description|linebreaks }}
```

Make sure the development server is running with the command `python manage.py runserver`. Now, open `http://127.0.0.1:8000/` in your browser and navigate to a product's detail page. It now contains a form to choose a quantity before adding the product to the cart. The page will look like this:



Choose a quantity and click on the Add to cart button. The form is submitted to the `cart_add` view via `POST`. The view adds the product to the cart in the session, including its current price and the selected quantity. Then, it redirects the user to the cart detail page, which will look like the following screenshot:



Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Red tea	2	Remove	$45.5	$91.0
Total					$91.0

Updating product quantities in the cart

When users see the cart, they might want to change product quantities before placing an order. We are going to allow users to change quantities from the cart detail page.

Edit the `views.py` file of the `cart` application and change the `cart_detail` view to this:

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
                          initial={'quantity': item['quantity'],
                          'update': True})
    return render(request, 'cart/detail.html', {'cart': cart})
```

We create an instance of `CartAddProductForm` for each item in the cart to allow changing product quantities. We initialize the form with the current item quantity and set the `update` field to `True` so that when we submit the form to the `cart_add` view, the current quantity is replaced with the new one.

Now, edit the `cart/detail.html` template of the `cart` application and find the following line:

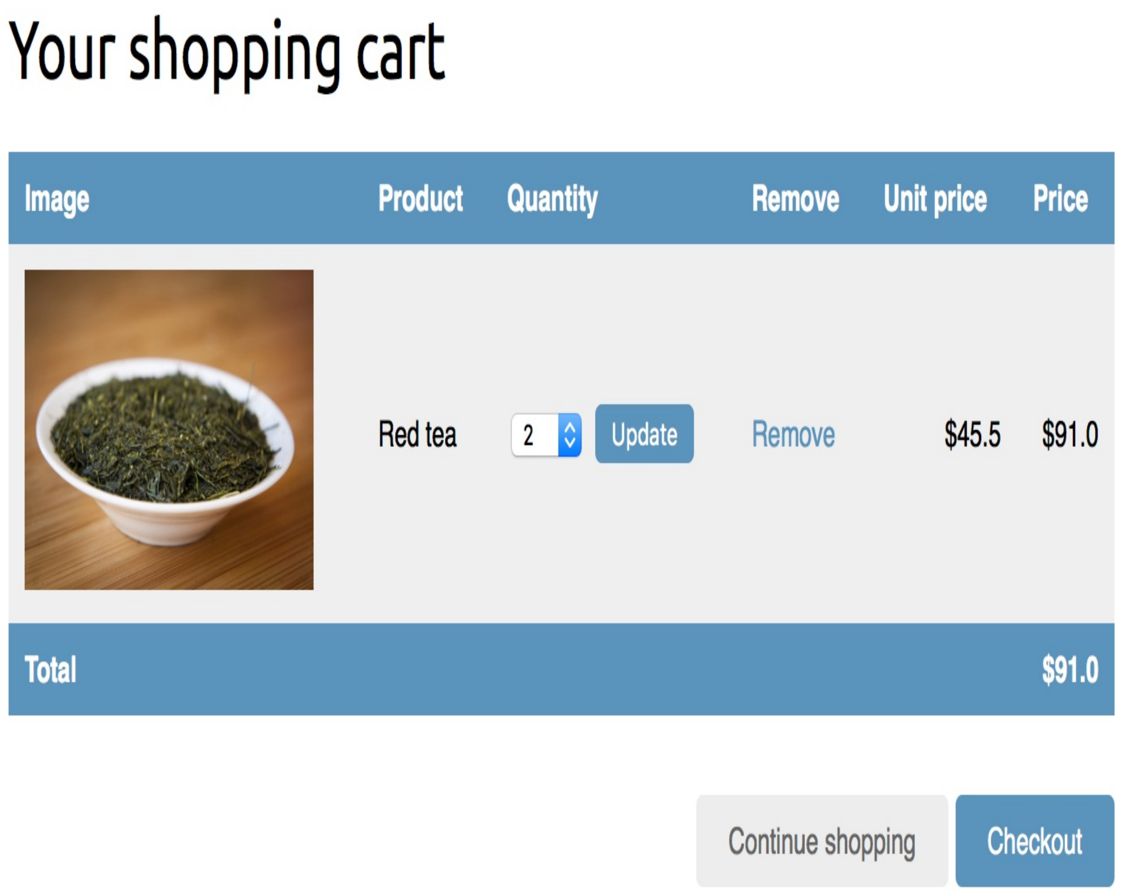
```
<td>{{ item.quantity }}</td>
```

Replace the previous line with the following code:

```
<td>
  <form action="{% url "cart:cart_add" product.id %}" method="post">
```

```
    {{ item.update_quantity_form.quantity }}
    {{ item.update_quantity_form.update }}
    <input type="submit" value="Update">
    {% csrf_token %}
  </form>
</td>
```

Open `http://127.0.0.1:8000/cart/` in your browser. You will see a form to edit the quantity for each cart item, shown as follows:



Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Red tea	2 Update	Remove	$45.5	$91.0
Total					$91.0

Change the quantity of an item and click on the Update button to test the new functionality. You can also remove an item from the cart by clicking the Remove link.

Creating a context processor for the current cart

You might have noticed that the message Your cart is empty is displayed in the header of the site, even when the cart contains items. We should display the total number of items in the cart and the total cost instead. Since this has to be displayed in all pages, we will build a context processor to include the current cart in the request context, regardless of the view that processes the request.

Context processors

A context processor is a Python function that takes the `request` object as an argument and returns a dictionary that gets added to the request context. They come in handy when you need to make something available globally to all templates.

By default, when you create a new project using the `startproject` command, your project contains the following template context processors, in the `context_processors` option inside the `TEMPLATES` setting:

- `django.template.context_processors.debug`: This sets the boolean `debug` and `sql_queries` variables in the context representing the list of SQL queries executed in the request.
- `django.template.context_processors.request`: This sets the `request` variable in the context.
- `django.contrib.auth.context_processors.auth`: This sets the `user` variable in the request.
- `django.contrib.messages.context_processors.messages`: This sets a `messages` variable in the context containing all messages that have been generated using the messages framework.

Django also enables `django.template.context_processors.csrf` to avoid cross-site request forgery attacks. This context processor is not present in the settings, but it is always enabled and cannot be turned off for security reasons.

You can see the list of all built-in context processors at `https://docs.dja` `ngoproject.com/en/2.0/ref/templates/api/#built-in-template-context-processors`.

Setting the cart into the request context

Let's create a context processor to set the current cart into the request context. We will be able to access the cart in any template.

Create a new file inside the `cart` application directory and name it `context_processors.py`. Context processors can reside anywhere in your code, but creating them here will keep your code well organized. Add the following code to the file:

```
from .cart import Cart

def cart(request):
    return {'cart': Cart(request)}
```

A context processor is a function that receives the `request` object as a parameter and returns a dictionary of objects that will be available to all the templates rendered using `RequestContext`. In our context processor, we instantiate the cart using the `request` object and make it available for the templates as a variable named `cart`.

Edit the `settings.py` file of your project and add `cart.context_processors.cart` to the `context_processors` option inside the `TEMPLATES` setting as follows:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # ...
                'cart.context_processors.cart',
            ],
        },
    },
]
```

The `cart` context processor will be executed every time a template is rendered using Django's `RequestContext`. The `cart` variable will be set in the context of your templates.

> *Context processors are executed in all the requests that use `RequestContext`. You might want to create a custom template tag instead of a context processor if your functionality is not needed in all templates, especially if it involves database queries.*

Now, edit the `shop/base.html` template of the `shop` application and find the following lines:

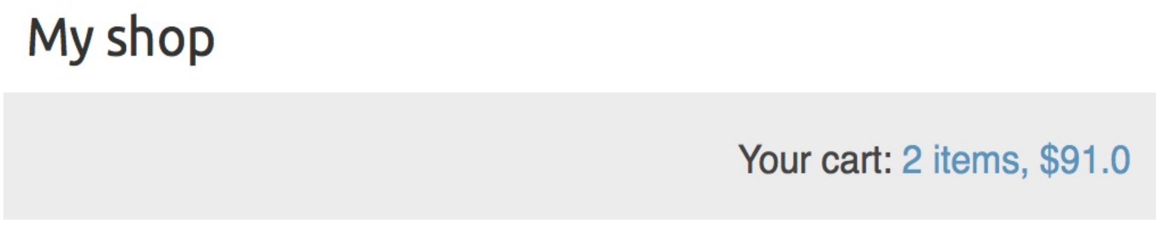
```
<div class="cart">
  Your cart is empty.
</div>
```

Replace the previous lines with the following code:

```
<div class="cart">
  {% with total_items=cart|length %}
    {% if cart|length > 0 %}
      Your cart:
      <a href="{% url "cart:cart_detail" %}">
        {{ total_items }} item{{ total_items|pluralize }},
        ${{ cart.get_total_price }}
      </a>
    {% else %}
      Your cart is empty.
    {% endif %}
  {% endwith %}
</div>
```

Reload your server using the command `python manage.py runserver`. Open `http://127.0.0.1:8000/` in your browser and add some products to the cart.
In the header of the website, you can see the total number of items

in the cart and the total cost, as follows:



My shop

Your cart: 2 items, $91.0

Registering customer orders

When a shopping cart is checked out, you need to save an order into the database. Orders will contain information about customers and the products they are buying.

Create a new application for managing customer orders using the following command:

```
python manage.py startapp orders
```

Edit the `settings.py` file of your project and add the new application to the `INSTALLED_APPS` setting as follows:

```
INSTALLED_APPS = [
    # ...
    'orders.apps.OrdersConfig',
]
```

You have activated the `orders` application.

Creating order models

You will need a model to store the order details, and a second model to store items bought, including their price and quantity. Edit the `models.py` file of the `orders` application and add the following code to it:

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

class Order(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField()
    address = models.CharField(max_length=250)
    postal_code = models.CharField(max_length=20)
    city = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    paid = models.BooleanField(default=False)

    class Meta:
        ordering = ('-created',)

    def __str__(self):
        return 'Order {}'.format(self.id)

    def get_total_cost(self):
        return sum(item.get_cost() for item in self.items.all())


class OrderItem(models.Model):
    order = models.ForeignKey(Order,
                              related_name='items',
                              on_delete=models.CASCADE)
    product = models.ForeignKey(Product,
                                related_name='order_items',
                                on_delete=models.CASCADE)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    quantity = models.PositiveIntegerField(default=1)

    def __str__(self):
        return '{}'.format(self.id)

    def get_cost(self):
        return self.price * self.quantity
```

The `Order` model contains several fields to store customer information and a `paid` boolean field, which defaults to `False`. Later on, we are going to use this field to differentiate between paid and unpaid orders. We also define a `get_total_cost()` method to obtain the total cost of the items bought in this order.

The `OrderItem` model allows us to store the product, quantity, and price paid for each item. We include `get_cost()` to return the cost of the item.

Run the next command to create initial migrations for the `orders` application:

```
python manage.py makemigrations
```

You will see the following output:

```
Migrations for 'orders':
  orders/migrations/0001_initial.py
    - Create model Order
    - Create model OrderItem
```

Run the following command to apply the new migration:

```
python manage.py migrate
```

Your order models are now synced to the database.

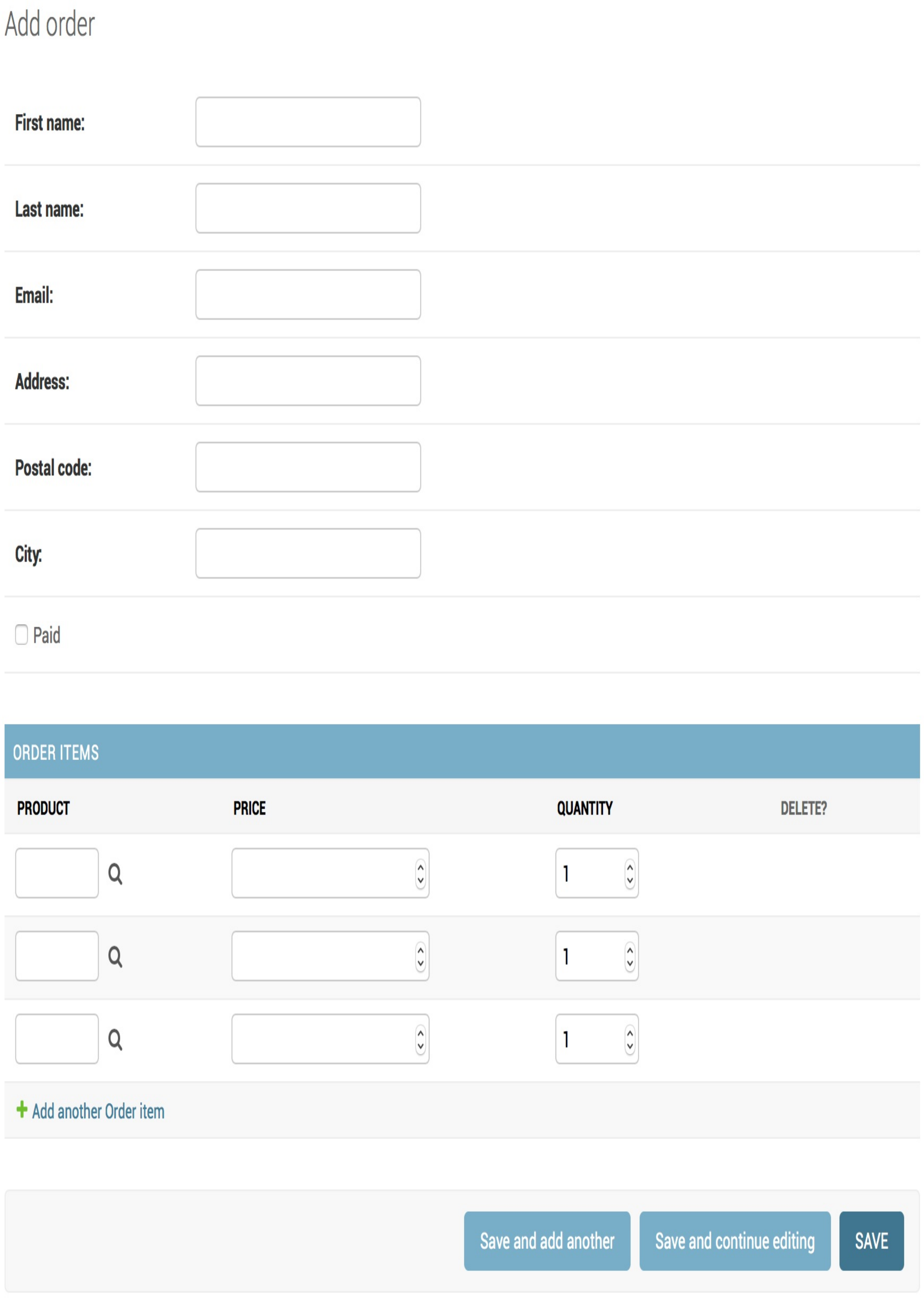
Including order models in the administration site

Let's add the order models to the administration site. Edit the `admin.py` file of the `orders` application to make it look like this:

```
from django.contrib import admin
from .models import Order, OrderItem

class OrderItemInline(admin.TabularInline):
    model = OrderItem
    raw_id_fields = ['product']

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                    'address', 'postal_code', 'city', 'paid',
                    'created', 'updated']
    list_filter = ['paid', 'created', 'updated']
    inlines = [OrderItemInline]
```

We use a `ModelInline` class for the `OrderItem` model to include it as an *inline* in the `OrderAdmin` class. An inline allows you to include a model on the same edit page its related model.

Run the development server with the command `python manage.py runserver`, and then open `http://127.0.0.1:8000/admin/orders/order/add/` in your browser. You will see the following page:



Add order

First name:	
Last name:	
Email:	
Address:	
Postal code:	
City:	

☐ Paid

ORDER ITEMS

PRODUCT	PRICE	QUANTITY	DELETE?
		1	
		1	
		1	

Creating customer orders

We will use the order models we created to persist the items contained in the shopping cart when the user finally places an order. A new order will be created following these steps:

1. Present users an order form to fill in their data
2. Create a new `Order` instance with the data entered, and create an associated `OrderItem` instance for each item in the cart
3. Clear all the cart content and redirect users to a success page

First, we need a form to enter the order details. Create a new file inside the `orders` application directory and name it `forms.py`. Add the following code to it:

```
from django import forms
from .models import Order

class OrderCreateForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address',
                  'postal_code', 'city']
```

This is the form that we are going to use for creating new `Order` objects. Now, we need a view to handle the form and create a new order. Edit the `views.py` file of the `orders` application and add the following code to it:

```
from django.shortcuts import render
from .models import OrderItem
from .forms import OrderCreateForm
from cart.cart import Cart
```

```
def order_create(request):
    cart = Cart(request)
    if request.method == 'POST':
        form = OrderCreateForm(request.POST)
        if form.is_valid():
            order = form.save()
            for item in cart:
                OrderItem.objects.create(order=order,
                                         product=item['product'],
                                         price=item['price'],
                                         quantity=item['quantity'])
            # clear the cart
            cart.clear()
            return render(request,
                          'orders/order/created.html',
                          {'order': order})
    else:
        form = OrderCreateForm()
    return render(request,
                  'orders/order/create.html',
                  {'cart': cart, 'form': form})
```

In the `order_create` view, we will obtain the current cart from the session with `cart = Cart(request)`. Depending on the request method, we will perform the following tasks:

- **GET request**: Instantiates the `OrderCreateForm` form and renders the `orders/order/create.html` template.
- **POST request**: Validates the data sent in the request. If the data is valid, we create a new order in the database using `order = form.save()`. We iterate over the cart items and create an `OrderItem` for each of them. Finally, we clear the cart content and render the template `orders/order/created.html`.

Create a new file inside the `orders` application directory and name it `urls.py`. Add the following code to it:

```
from django.urls import path
from . import views

app_name = 'orders'

urlpatterns = [
    path('create/', views.order_create, name='order_create'),
]
```

This is the URL pattern for the `order_create` view. Edit the `urls.py` file of `myshop` and include the following pattern. Remember to place it before the `shop.urls` pattern:

```
path('orders/', include('orders.urls', namespace='orders')),
```

Edit the `cart/detail.html` template of the `cart` application and edit this line:

```
<a href="#" class="button">Checkout</a>
```

Add the `order_create` URL as follows:

```
<a href="{% url "orders:order_create" %}" class="button">
  Checkout
</a>
```

Users can now navigate from the cart detail page to the order form. We still need to define templates for placing orders. Create the following file structure inside the `orders` application directory:

```
templates/
    orders/
        order/
            create.html
            created.html
```

Edit the `orders/order/create.html` template and include the following code:

```
{% extends "shop/base.html" %}

{% block title %}
  Checkout
{% endblock %}

{% block content %}
  <h1>Checkout</h1>

  <div class="order-info">
    <h3>Your order</h3>
    <ul>
      {% for item in cart %}
        <li>
          {{ item.quantity }}x {{ item.product.name }}
          <span>${{ item.total_price }}</span>
        </li>
      {% endfor %}
    </ul>
    <p>Total: ${{ cart.get_total_price }}</p>
  </div>

  <form action="." method="post" class="order-form">
    {{ form.as_p }}
    <p><input type="submit" value="Place order"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

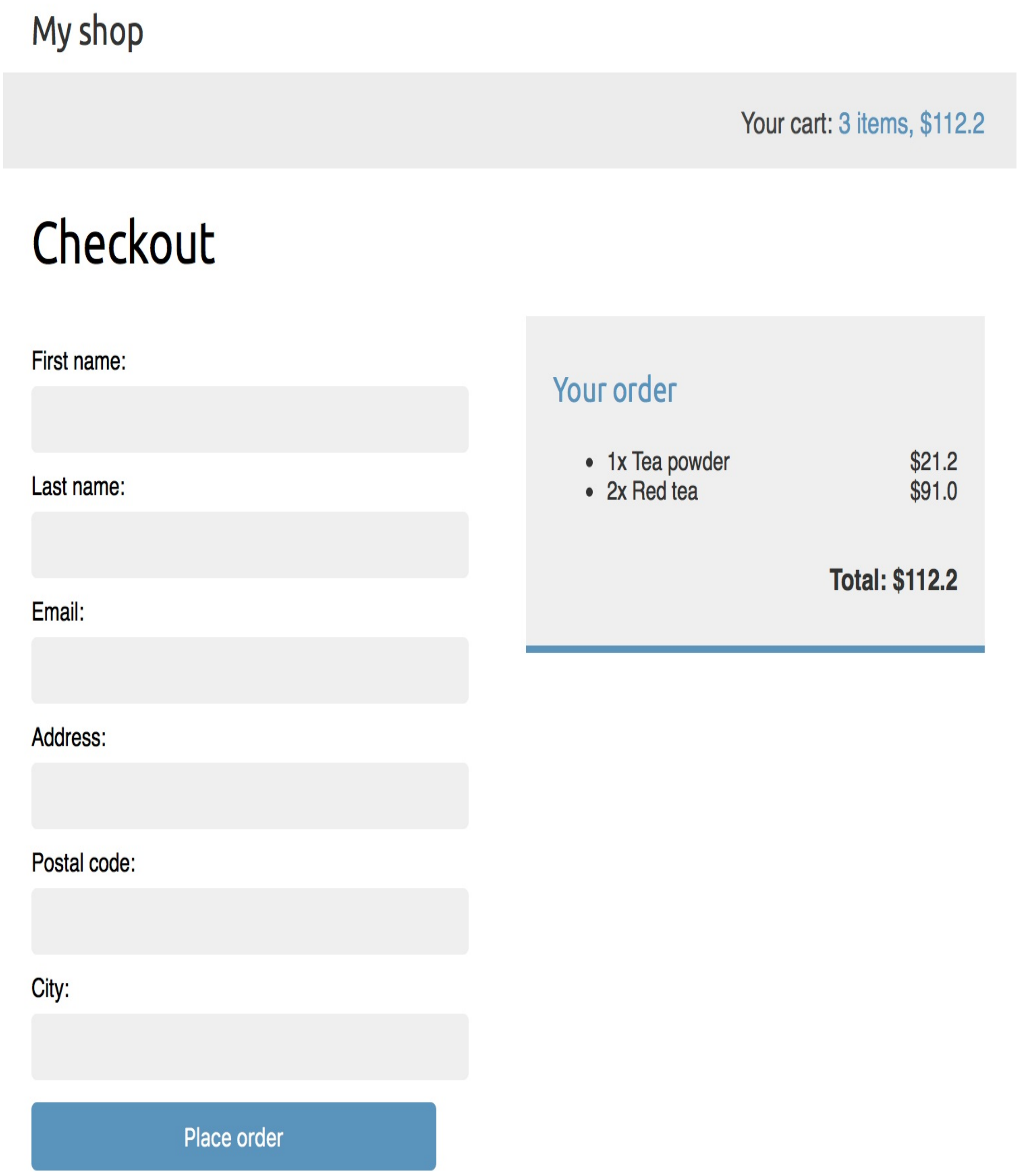
This template displays the cart items, including totals, and the form to place an order.

Edit the `orders/order/created.html` template and add the following code:

```
{% extends "shop/base.html" %}

{% block title %}
  Thank you
{% endblock %}

{% block content %}
  <h1>Thank you</h1>
  <p>Your order has been successfully completed. Your order number is
  <strong>{{ order.id }}</strong>.</p>
{% endblock %}
```

This is the template that we render when the order is successfully created.

Start the web development server to track new files. Open `http://127.0.0.1:8000/` in your browser, add a couple of products to the cart, and continue to the checkout page. You will see a page like the one following:



My shop

Your cart: 3 items, $112.2

Checkout

First name:

Last name:

Email:

Address:

Postal code:

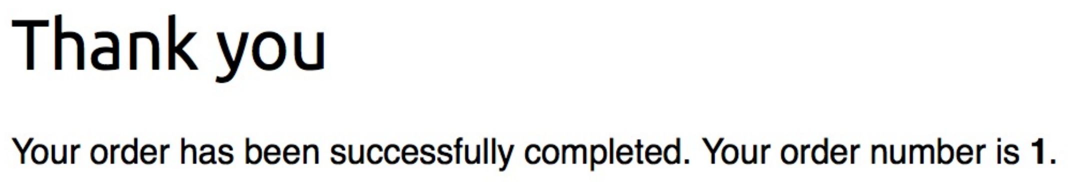
City:

Your order

1x Tea powder	$21.2
2x Red tea	$91.0

Total: $112.2

Fill in the form with the valid data and click on the Place order button. The order will be created and you will see a success page like this:


Thank you

Your order has been successfully completed. Your order number is **1**.

Now, go and check the orders in administration site .