8

Building an Online Shop

In the previous chapter, you created a follow system and built a user activity stream. You also learned how Django signals work and integrated Redis into your project to count image views.

In this chapter, you will start a new Django project that consists of a fully featured online shop. This chapter and the following two chapters will show you how to build the essential functionalities of an e-commerce platform. Your online shop will enable clients to browse products, add them to the cart, apply discount codes, go through the checkout process, pay with a credit card, and obtain an invoice. You will also implement a recommendation engine to recommend products to your customers, and you will use internationalization to offer your site in multiple languages.

In this chapter, you will learn how to:

- Create a product catalog
- Build a shopping cart using Django sessions
- Create custom template context processors
- Manage customer orders
- Configure Celery in your project with RabbitMQ as a message broker
- Send asynchronous notifications to customers using Celery
- Monitor Celery using Flower

Functional overview

Figure 8.1 shows a representation of the views, templates, and main functionalities that will be built in this chapter:

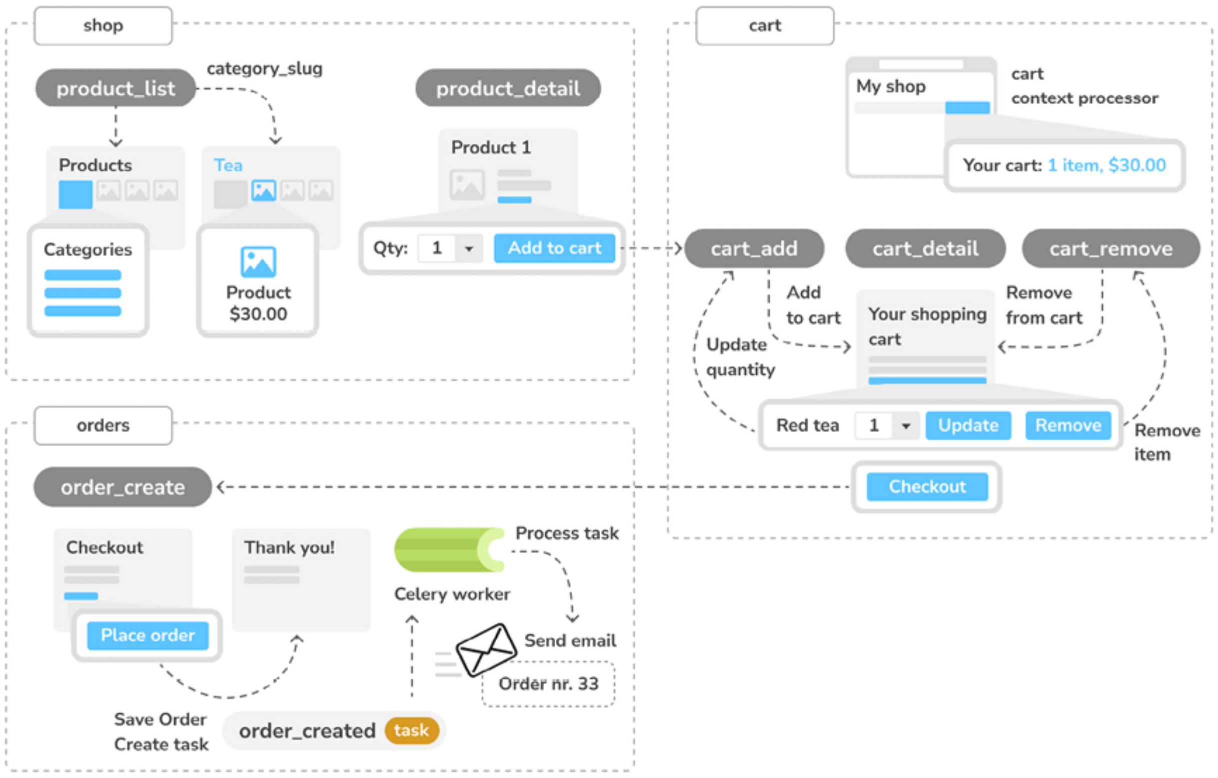


Figure 8.1: Diagram of functionalities built in Chapter 8

In this chapter, you will implement the `product_list` view to list all products and the `product_detail` view to display a single product. You will allow filtering products by category in the `product_list` view using the `category_slug` parameter. You will implement a shopping cart using sessions and you will build the `cart_detail` view to display the cart items. You will create the `cart_add` view to add products to the cart and update quantities, and the `cart_remove` view to remove products from the cart. You will implement the `cart` template context processor to display the number of cart items and total cost on the site header. You will also create the `order_create` view to place orders, and you will use Celery to implement the `order_created` asynchronous task that sends out an email confirmation to clients when they place an order. This chapter will provide you with the knowledge to implement user sessions in your application and show you how to work with asynchronous tasks. Both are very common use cases that you can apply to almost any project.

The source code for this chapter can be found at https://github.com/PacktPublishing/Django-5-by-example/tree/main/Chapter08.

All Python modules used in this chapter are included in the `requirements.txt` file in the source code that comes along with this chapter. You can follow the instructions to install each Python module below or you can install all requirements at once with the command `python -m pip install -r requirements.txt`.

Creating an online shop project

Let's start with a new Django project to build an online shop. Your 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 and use the following command to create a new virtual environment for this project within the `env/` directory:

```
python -m venv env/myshop
```

If you are using Linux or macOS, run the following command to activate your virtual environment:

```
source env/myshop/bin/activate
```

If you are using Windows, use the following command instead:

```
.\env\myshop\Scripts\activate
```

The shell prompt will display your active virtual environment, as follows:

```
(myshop)laptop:~ zenx$
```

Install Django in your virtual environment with the following command:

```
python -m pip install Django~=5.0.4
```

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

```
django-admin startproject myshop
```

The initial project structure has been created. Use the following commands to get into your project directory and create a new application named `shop`:

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

Edit `settings.py` and add the following line highlighted in bold to the `INSTALLED_APPS` list:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
 '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 your shop will consist of products that are organized into different categories. Each product will have a name, an optional description, an optional image, a price, and its 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)
    slug = models.SlugField(max_length=200, unique=True)
 class Meta:
        ordering = ['name']
        indexes = [
            models.Index(fields=['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)
    slug = models.SlugField(max_length=200)
    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']
        indexes = [
            models.Index(fields=['id', 'slug']),
            models.Index(fields=['name']),
            models.Index(fields=['-created']),
        ]
 def __str__(self):
 return self.name
```

These are the `Category` and `Product` models. The `Category` model consists of a `name` field and a unique `slug` field (`unique` implies the creation of an index). In the `Meta` class of the `Category` model, we have defined an index for the `name` field.

The `Product` model fields are as follows:

- `category`: A `ForeignKey` to the `Category` model. This is a one-to-many 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 have defined a multiple-field index for the `id` and `slug` fields. Both fields are indexed together to improve performance for queries that utilize the two fields.

We plan to query products by both `id` and `slug`. We have added an index for the `name` field and an index for the `created` field. We have used a hyphen before the field name to define the index in descending order.

Figure 8.2 shows the two data models you have created:

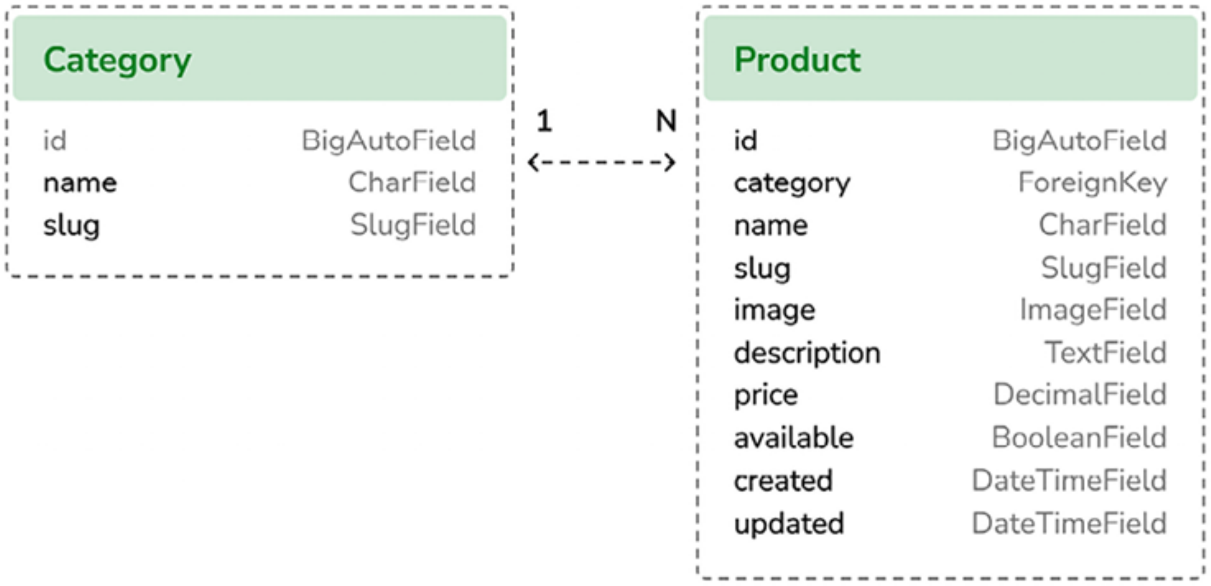


Figure 8.2: Models for the product catalog

In *Figure 8.2*, you can see the different fields of the data models and the one-to-many relationship between the `Category` and the `Product` models.

These models will result in the following database tables displayed in *Figure 8.3*:

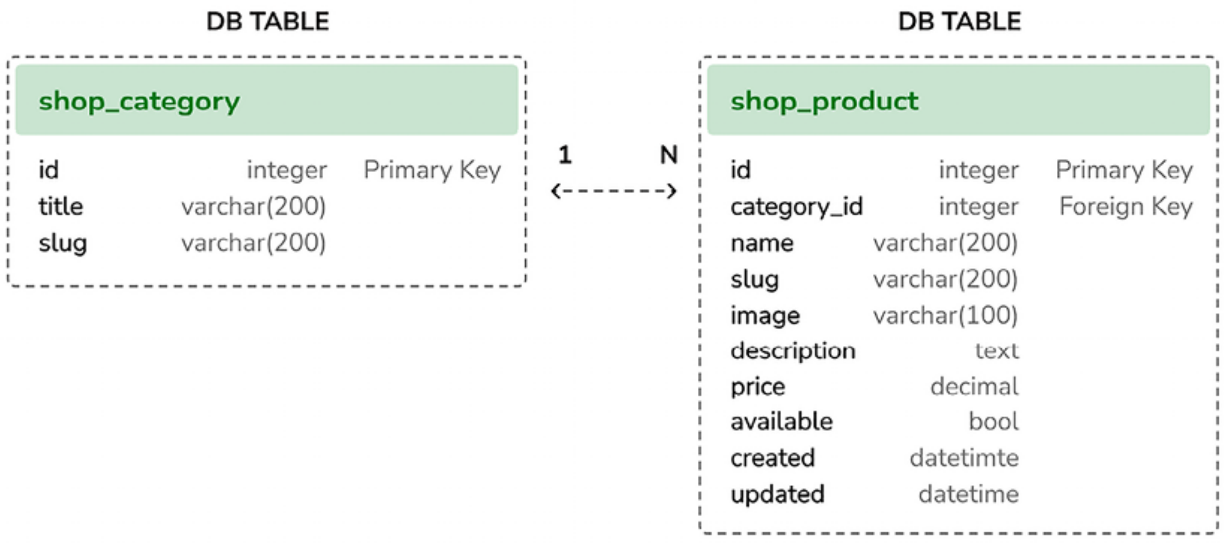


Figure 8.3: Database tables for the product catalog models

The one-to-many relationship between both tables is defined with the `category_id` field in the `shop_product` table, which is used to store the ID of the related `Category` for each `Product` object.

Let's create the initial database migrations for the `shop` application. Since you are going to deal with images in your models, you will need to install the Pillow library. Remember that in *Chapter 4*, *Building a Social Website*, you learned how to install the Pillow library to manage images. Open the shell and install `Pillow` with the following command:

```
python -m pip install Pillow==10.3.0
```

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
```

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 administration site

Let’s add your models to the administration site so that you 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 you 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.

You 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.

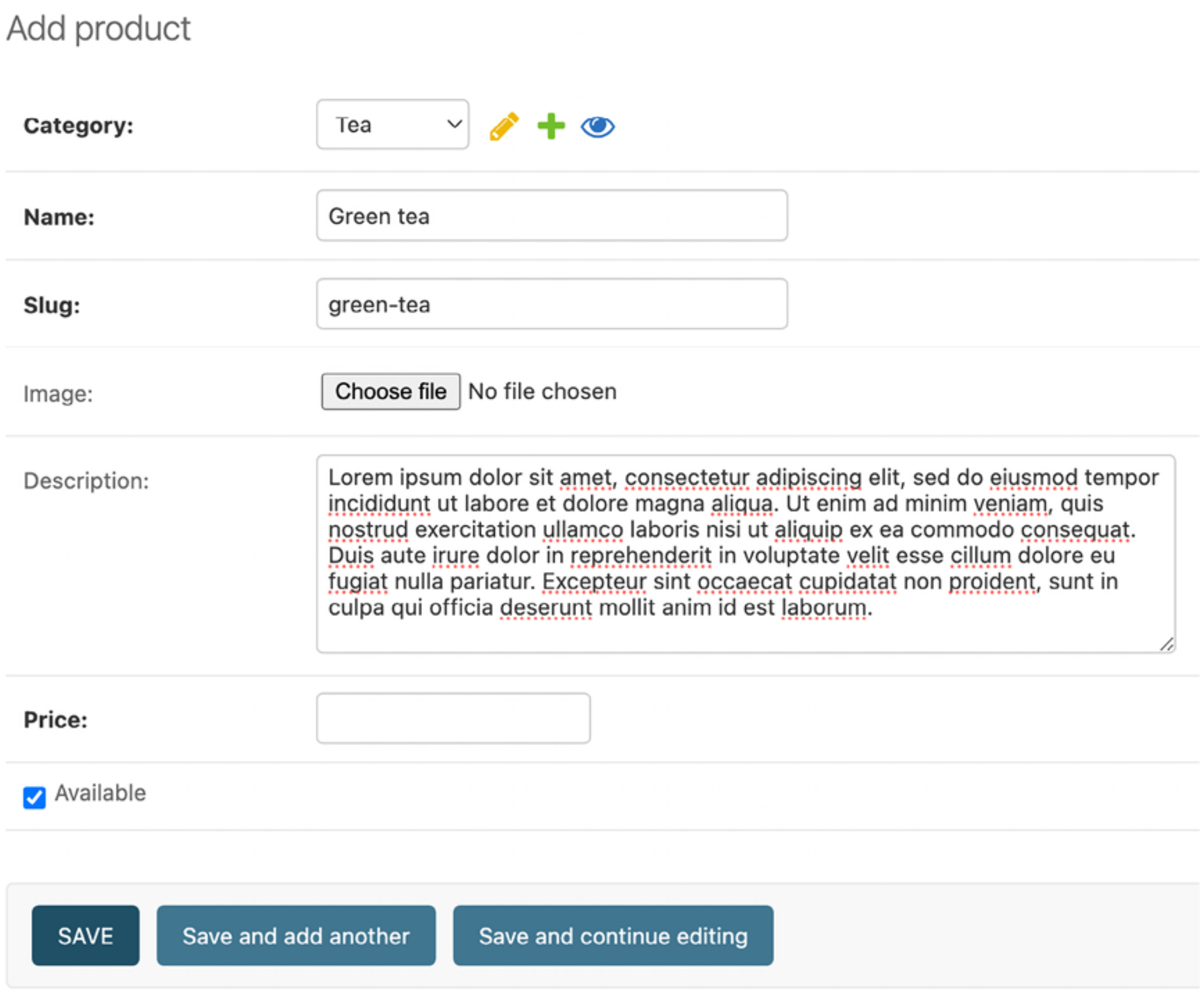
Now create a superuser for your site using the following command:

```
python manage.py createsuperuser
```

Enter the desired username, email, and password. Run the development server with the following 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 **Add product** form should look as follows:

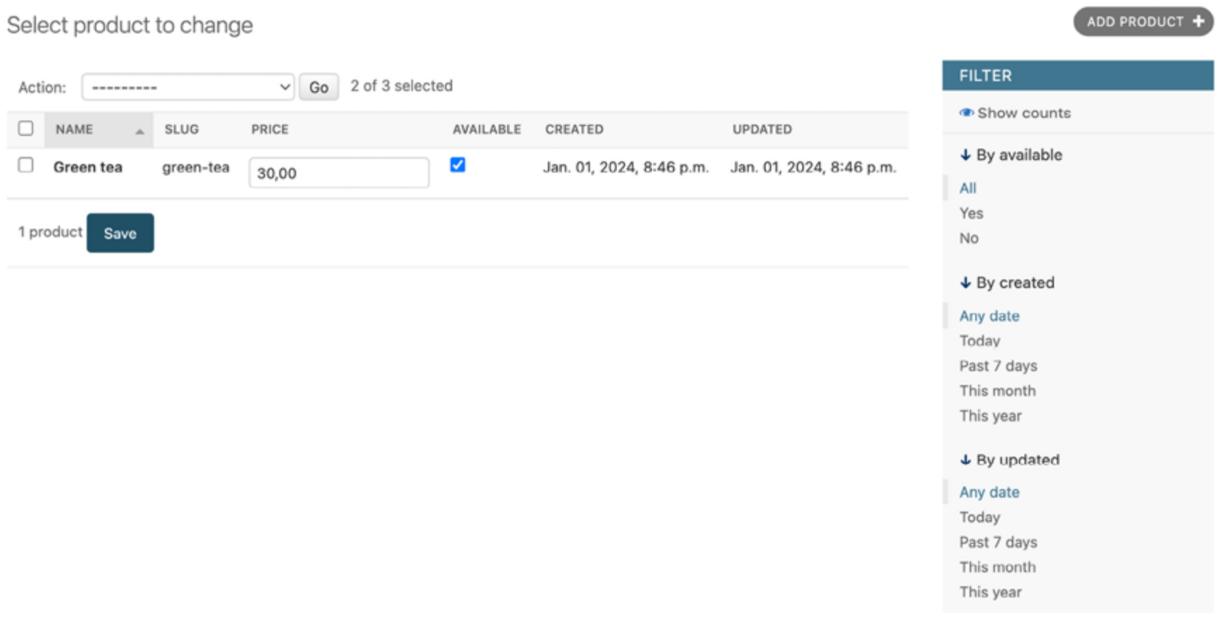


Add product

Category:	Tea
Name:	Green tea
Slug:	green-tea
Image:	Choose file No file chosen
Description:	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.
Price:	
✓ Available	

Figure 8.4: The product creation form

Click on the **SAVE** button. The product change list page of the administration page will then look like this:


Select product to change

ADD PRODUCT

Action: --------- Go 2 of 3 selected

	NAME	SLUG	PRICE	AVAILABLE	CREATED	UPDATED
☐	Green tea	green-tea	30,00	✓	Jan. 01, 2024, 8:46 p.m.	Jan. 01, 2024, 8:46 p.m.

1 product Save

Figure 8.5: The product change list page

Building catalog views

In order to display the product catalog, you 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 highlighted in bold:

```
from django.shortcuts import get_object_or_404, render
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_slu
        products = products.filter(category=category)
 return render(
 request,
 'shop/product/list.html',
 {
 'category': category,
 'categories': categories,
 'products': products
 }
     )
```

In the preceding code, you filter the `QuerySet` with `available=True` to retrieve only available products. You use an optional `category_slug` parameter to optionally filter products by a given category.

You 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. You can get this instance just through the ID since it's a unique attribute. However, you include the slug in the URL to build SEO-friendly URLs for products.

After building the product list and detail views, you 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 your product catalog. You 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. You 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 include, path
    urlpatterns = [
        path('admin/', admin.site.urls),
        path('', include('shop.urls', namespace='shop')),
    ]
```

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

Next, 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. The new code is highlighted in bold:

```
from django.db import models
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, you use the URL patterns that you just defined in the `urls.py` file.

Creating catalog templates

Now you 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
```

You 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 you will use for your shop. In order to include the CSS styles and images that are used by the templates, you need to copy the static files that accompany this chapter, which are located in the `static/` directory of the `shop` application. Copy them to the same location in your project. You can find the contents of the directory at https://github.com/PacktPublishing/Django-5-by-Example/tree/main/Chapter08/myshop/shop/static.

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 %
</a>
<a href="{{ product.get_absolute_url }}">{{ product.name }}</a>
<br>
        ${{ product.price }}
      </div>
    {% endfor %}
  </div>
{% endblock %}
```

Make sure that no template tag is split across multiple lines.

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 category. Since the `image` field of the `Product` model can be blank, you need to provide a default image for the products that don't have an image. The image is located in your static files directory with the relative path `img/no_image.png`.

Since you are using `ImageField` to store product images, you 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 = 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 you 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 highlighted in bold:

```
from django.conf import settings
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import include, path
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('shop.urls', namespace='shop')),
]
if settings.DEBUG:
    urlpatterns += static(
        settings.MEDIA_URL, document_root=settings.MEDIA_ROOT
    )
```

Remember that you only serve static files this way during development. In a production environment, you should never serve static files with Django; the Django development server doesn't serve static files in an efficient manner. *Chapter 17*, *Going Live*, will teach you how to serve static files in a production environment.

Run the development server with the following command:

```
python manage.py runserver
```

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 will look similar to this:

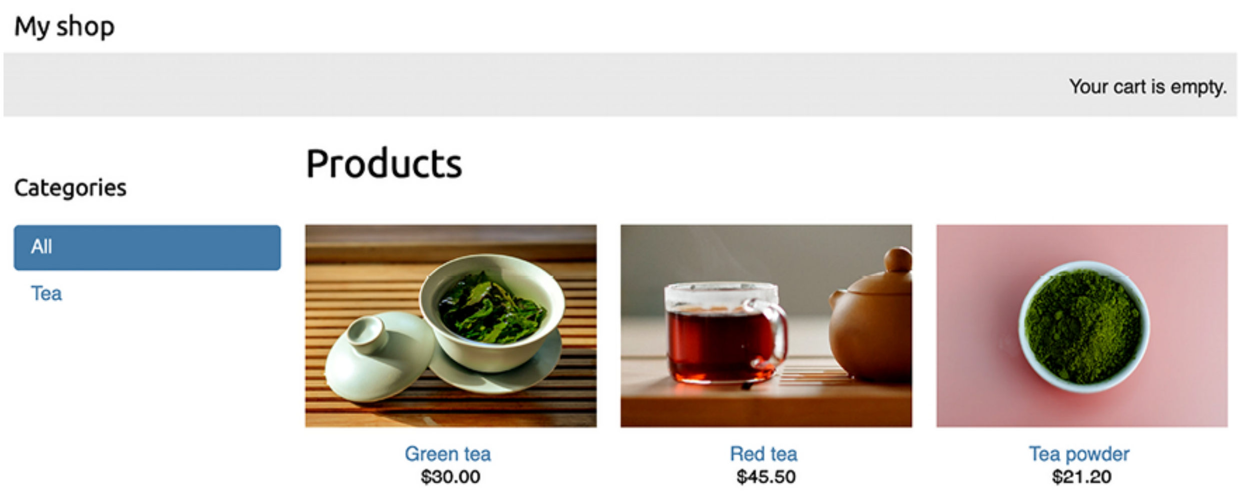


Figure 8.6: The product list page

Credits for images in this chapter:

- *Green tea*: Photo by Jia Ye on Unsplash
- *Red tea*: Photo by Manki Kim on Unsplash
- *Tea powder*: Photo by Phuong Nguyen on Unsplash

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

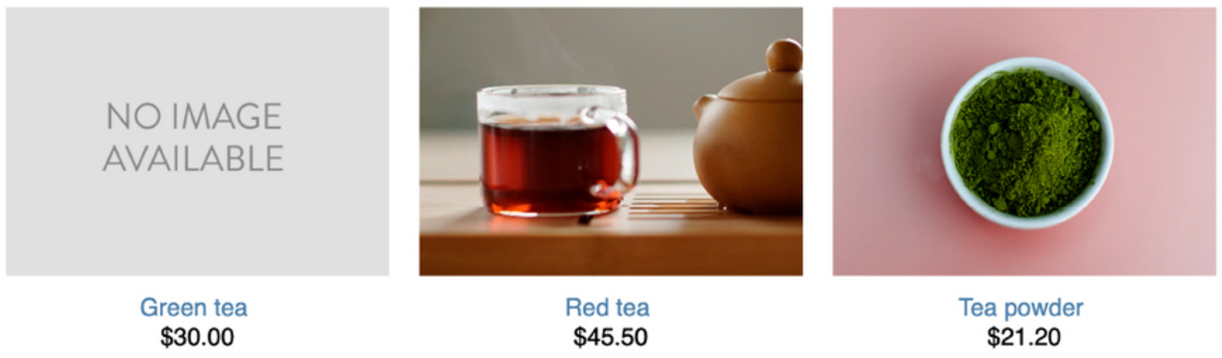


Figure 8.7: The product list displaying a default image for products that have no image

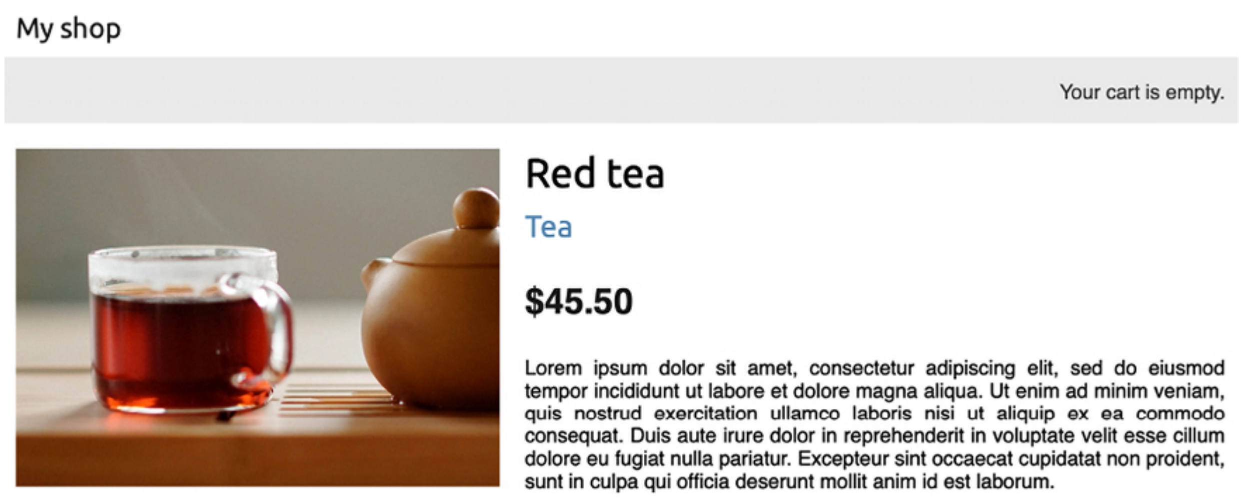
Edit the `shop/product/detail.html` template and add the following code to it:

```
{% 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 %}
```

In the preceding code, you 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:



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.

Figure 8.8: The product detail page

You have now created a basic product catalog. Next, you will implement a shopping cart that allows users to add any product to it while browsing the online shop.

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 amount 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 a user's visit.

You 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 the cart. You 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 other 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'
```

You can retrieve a session key as follows:

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

You can delete a key you previously stored in the session as follows:

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

When users log in to the site, their anonymous session is lost, and a new session is created for 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. You can do this by retrieving the session data before you log in the user using the `login()` function of the Django authentication system and storing it in the session after that.

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_HTTPONLY`: Whether to use the `HttpOnly` flag on the session cookie. If this is set to `True`, client-side JavaScript will not be

able to access the session cookie. The default value is `True` for increased security against user session hijacking.

- `SESSION_COOKIE_SECURE`: A Boolean indicating that the cookie should only be sent if the connection is an HTTPS connection. The default value is `False`.
- `SESSION_EXPIRE_AT_BROWSER_CLOSE`: A Boolean indicating that the session has to expire when the browser is closed. The default value is `False`.
- `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. The default value is `False`.

You can see all the session settings and their default values at https://docs.djangoproject.com/en/5.0/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

You 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
- The quantity selected for the product
- The unit price for the product

Since product prices may vary, let's take the approach of storing the product's price along with the product itself when it's added to the cart. By doing so, you use the current price of the product when users add it to their cart, no matter whether the product's price is changed afterward. This means that the price that the item has when the client adds it to the cart is maintained for that client in the session until checkout is completed or the session finishes.

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

- When a cart is needed, you check whether a custom session key is set. If no cart is set in the session, you create a new cart and save it in the cart session key.
- For successive requests, you perform the same check and get the cart items from the cart session key. You 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 you are going to use to store the cart in the user session. Since Django sessions are managed per visitor, you 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 with the following line highlighted in bold:

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

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:
    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 you to manage the shopping cart. You require the cart to be initialized with a `request` object. You store the current session using `self.session = request.session` to make it accessible to the other methods of the `Cart` class.

First, you 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, you create an empty cart by setting an empty dictionary in the session.

You will build your `cart` dictionary with product IDs as keys, and for each product key, a dictionary will be a value that includes quantity and price. By doing this, you can guarantee that a product will not be added more than once to the cart. This way, you can also simplify retrieving 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:
    # ...
 def add(self, product, quantity=1, override_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 override_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`.
- `override_quantity`: A Boolean that indicates whether the quantity needs to be overridden with the given quantity (`True`) or whether the new quantity has to be added to the existing quantity (`False`).

You use the product ID as a key in the cart's content dictionary. You 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 you persist is a dictionary with quantity and price figures for the product. The product's price is converted from decimal into a string to serialize it. Finally, you 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.

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

```
class Cart:
    # ...
```

```
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.

You 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:
    # ...
 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, you retrieve the `Product` instances that are present in the cart to include them in the cart items. You copy the current cart in the `cart` variable and add the `Product` instances to it. Finally, you iterate over the cart items, converting each item's price back into decimal, and adding a `total_price` attribute to each item. This `__iter__()` method will allow you to easily iterate over the items in the cart in views and templates.

You also need a way to return the total number of items in the cart. When the `len()` function is executed on an object, Python calls its `__len__()` method to retrieve its length. Next, you 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:
    # ...
 def __len__(self):
 """
        Count all items in the cart.
        """
 return sum(item['quantity'] for item in self.cart.values())
```

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

Finally, add a method to clear the cart session:

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

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

Creating shopping cart views

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

- A view to add or update items in the cart that 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

To add items to the cart, you 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
    )
    override = forms.BooleanField(
        required=False,
        initial=False,
        widget=forms.HiddenInput
    )
```

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

- `quantity`: This allows the user to select a quantity between 1 and 20. You use a `TypedChoiceField` field with `coerce=int` to convert the input into an integer.
- `override`: 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 overridden with the given quantity (`True`). You use a `HiddenInput` widget for this field since you 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 highlighted in bold:

```
from django.shortcuts import get_object_or_404, redirect, render
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'],
            override_quantity=cd['override']
        )
    return redirect('cart:cart_detail')
```

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

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

```
@require_POST
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. You use the `require_POST` decorator to allow only `POST` requests. You retrieve the `Product` instance with the given ID and remove the product from the cart. Then, you redirect the user to the `cart_detail` URL.

Finally, you 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.

You have created views to add items to the cart, update quantities, remove items from the cart, and display the cart's contents. 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 URL patterns 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 highlighted in bold 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 you 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" %}{% endi
</a>
```

```
</td>
<td>{{ product.name }}</td>
<td>{{ item.quantity }}</td>
<td>
<form action="{% url "cart:cart_remove" product.id %}" method="p
<input type="submit" value="Remove">
                {% csrf_token %}
              </form>
</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 %}
```

Make sure that no template tag is split across multiple lines.

This is the template that is used to display the cart's contents. It contains a table with the items stored in the current cart. You allow users to change the quantity of the selected products using a form that is posted to the `cart_add` view. You also allow users to remove items from the cart by providing a **Remove** button for each of them. Finally, you use an HTML form with an `action` attribute that points to the `cart_remove` URL including the product ID.

Adding products to the cart

Now you 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_f
    )
```

Edit the `shop/product/detail.html` template of the `shop` application and add the following form to the product price, as follows. New lines are highlighted in bold:

```
...
<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 }}
...
```

Run the development server with the following 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 will contain a form to choose a quantity before adding the product to the cart. The page will look like this:

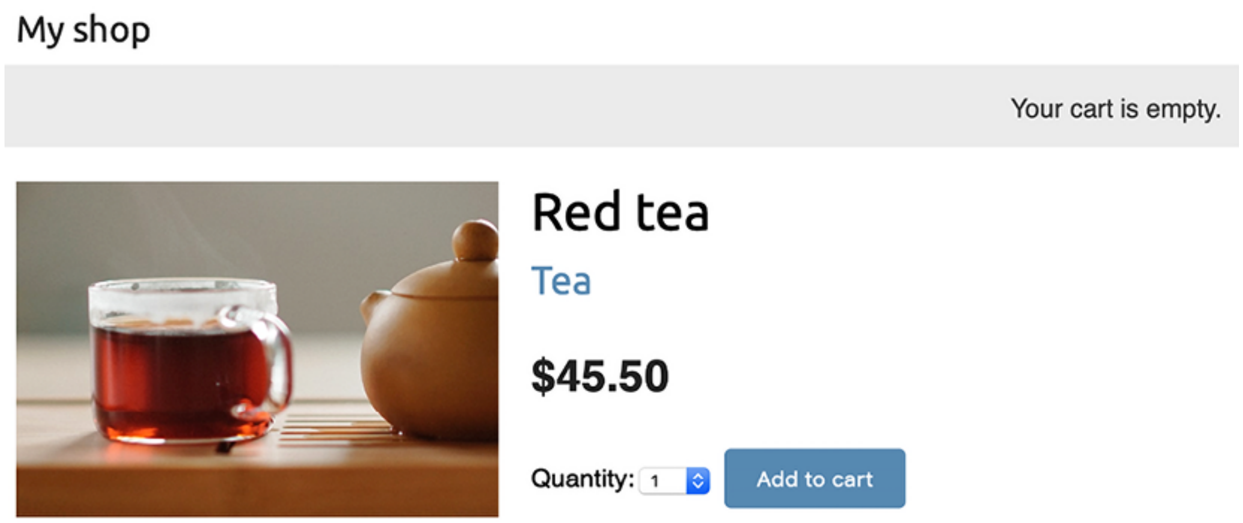


Figure 8.9: The product detail page, including the Add to cart button

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 *Figure 8.10*:

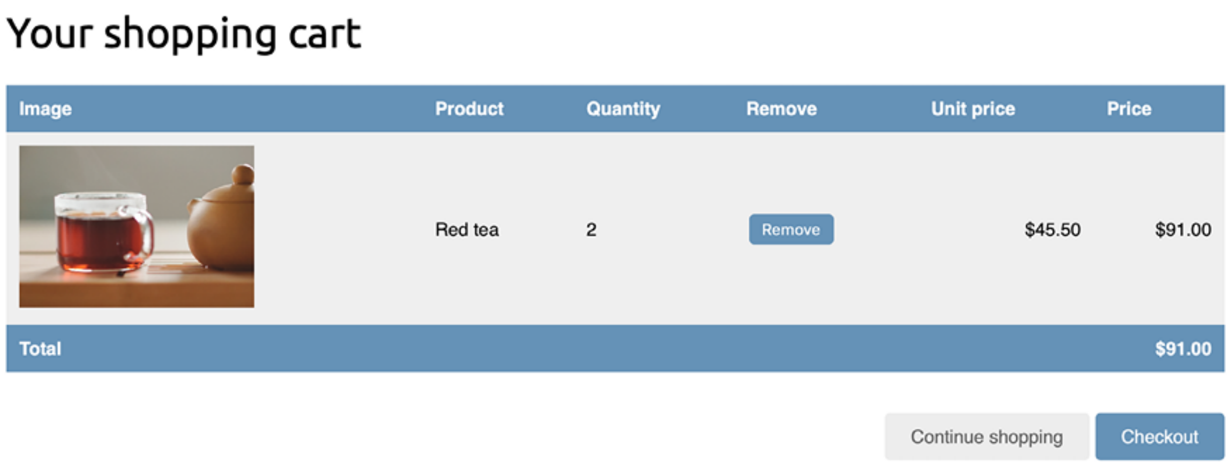


Figure 8.10: The cart detail page

Updating product quantities in the cart

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

Edit the `views.py` file of the `cart` application and add the following lines highlighted in bold to the `cart_detail` view:

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
            initial={'quantity': item['quantity'], 'override': T
        )
    return render(request, 'cart/detail.html', {'cart': cart})
```

You create an instance of `CartAddProductForm` for each item in the cart to allow changing product quantities. You initialize the form with the current item quantity and set the `override` field to `True` so that when you 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:

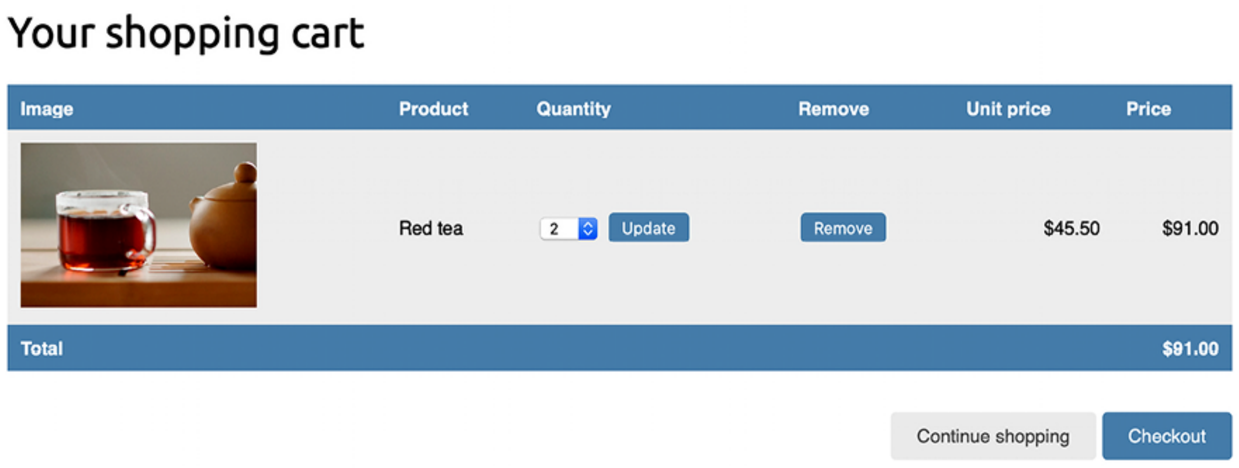
```
<td>
 <form action="{% url "cart:cart_add" product.id %}" method="pos
    {{ item.update_quantity_form.quantity }}
    {{ item.update_quantity_form.override }}
 <input type="submit" value="Update">
    {% csrf_token %}
 </form>
</td>
```

Run the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/cart/` in your browser.

You will see a form to edit the quantity for each cart item, as follows:



Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Red tea	2 Update	Remove	$45.50	$91.00
Total					$91.00

Figure 8.11: The cart detail page, including the form to update product quantities

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** button.

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. You should display the total number of items in the cart and the total cost instead. Since this has to be displayed on all pages, you need to 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.

Context processors 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 the messages that have been generated using the messages framework.

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

You can see the list of all built-in context processors at https://docs.djangoproject.com/en/5.0/ref/templates/api/#built-in-template-context-processors.

Setting the cart in the request context

Let’s create a context processor to set the current cart in the request context. With it, you 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)}
```

In your context processor, you 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. The new line is highlighted in bold:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTempla
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.mess
 '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. You can read more about `RequestContext` at https://docs.djangoproject.com/en/5.0/ref/templates/api/#django.template.RequestContext.

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.

Next, 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:

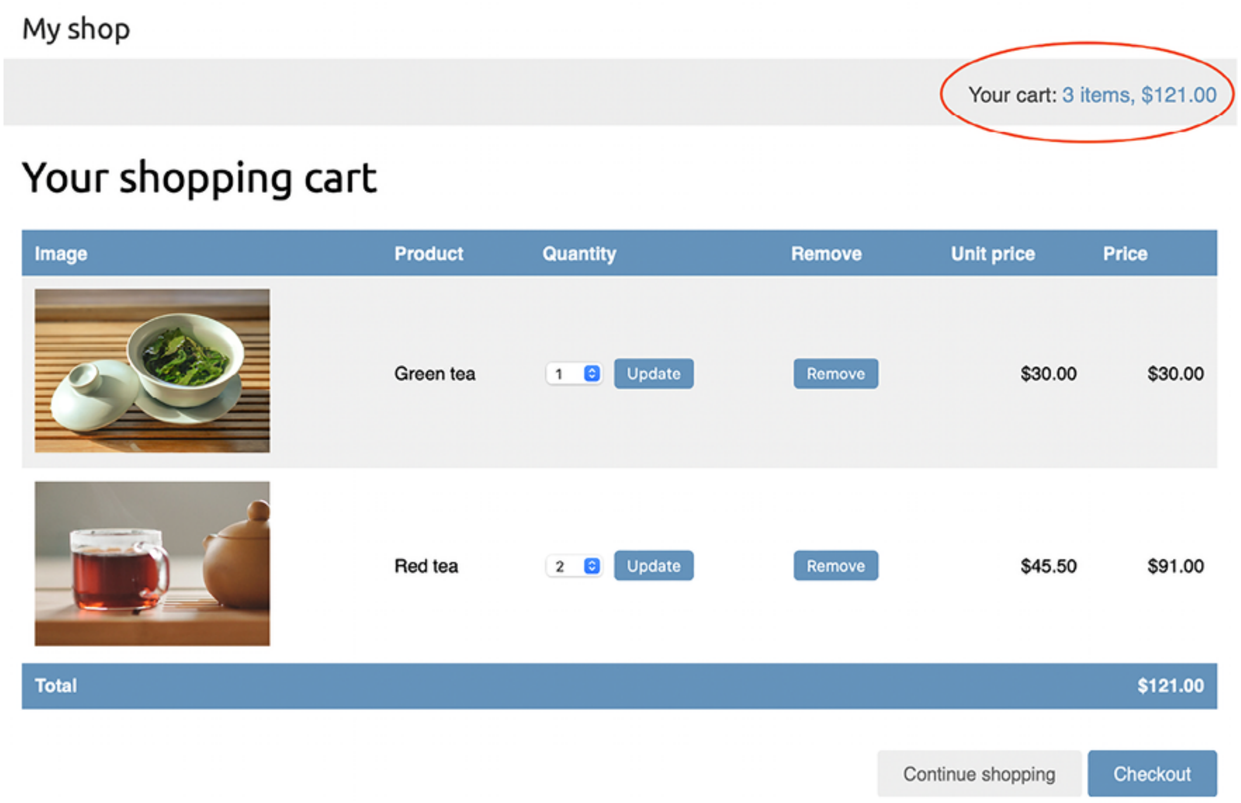
```
<div class="cart">
  {% with total_items=cart|length %}
    {% if total_items > 0 %}
      Your cart:
 <a href="{% url "cart:cart_detail" %}">
        {{ total_items }} item{{ total_items|pluralize }},
        ${{ cart.get_total_price }}
 </a>
    {% else %}
      Your cart is empty.
    {% endif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

Restart the development server with the following 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 now see the total number of items in the cart and the total cost, as follows:


My shop

Your cart: 3 items, $121.00

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Green tea	1 Update	Remove	$30.00	$30.00
	Red tea	2 Update	Remove	$45.50	$91.00
Total					$121.00

Figure 8.12: The site header displaying the current items in the cart

Congratulations! You have completed the shopping cart functionality. This is a significant milestone in your online shop project. Next, you are going to create the functionality to register customer orders, which is another foundational element of any e-commerce platform.

Registering customer orders

When a shopping cart is checked out, you need to save an order in 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 = [
    # ...
'cart.apps.CartConfig',
 'orders.apps.OrdersConfig',
'shop.apps.ShopConfig',
]
```

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
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']
        indexes = [
            models.Index(fields=['-created']),
        ]
    def __str__(self):
        return f'Order {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(
        'shop.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 str(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, you are going to use this field to differentiate between paid and unpaid orders. We have also defined a `get_total_cost()` method to obtain the total cost of the items bought in this order.

The `OrderItem` model allows you to store the product, quantity, and price paid for each item. We have defined a `get_cost()` method that returns the cost of the item by multiplying the item price with the quantity. In the `product` field, we use the string `'shop.Product'` with the format `app.Model`, which is another way to point to related models and also a good method to avoid circular imports.

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

```
python manage.py makemigrations
```

You will see output similar to the following:

```
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
```

You will see the following output:

```
Applying orders.0001_initial... OK
```

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 and add the following code highlighted in bold:

```
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]
```

You 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 as its related model.

Run the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/orders/order/add/` in your browser. You will see the following page:

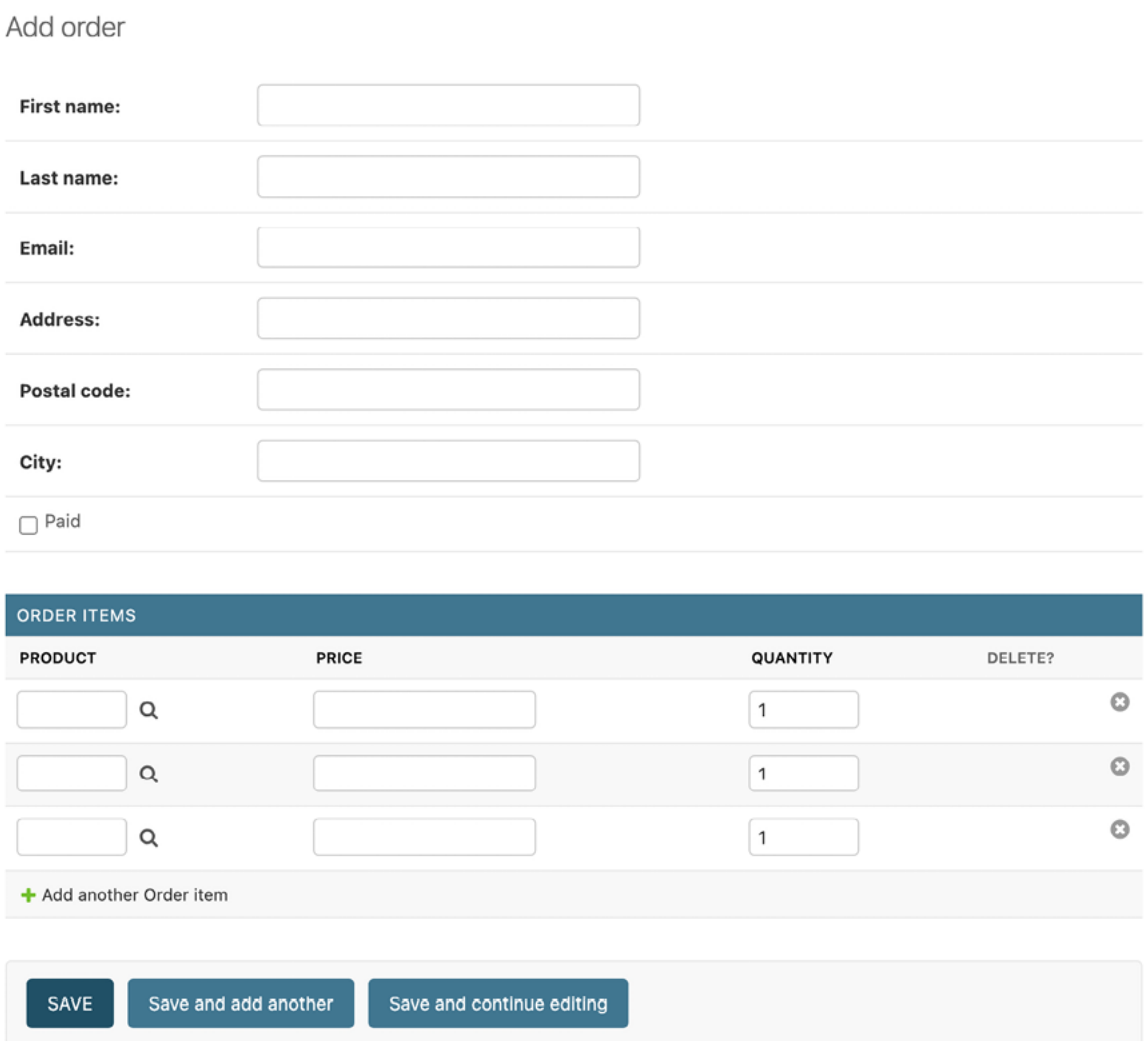


Figure 8.13: The Add order form, including OrderItemInline

Creating customer orders

You will use the order models that you 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 a user with 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's contents and redirect the user to a success page.

First, you 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 you are going to use to create new `Order` objects. Now you 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 highlighted in bold:

```
from cart.cart import Cart
from django.shortcuts import render
from .forms import OrderCreateForm
from .models import OrderItem
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':
            )
 else:
        form = OrderCreateForm()
 return render(
        request,
 'orders/order/create.html',
        {'cart': cart, 'form': form}
    )
```

In the `order_create` view, you obtain the current cart from the session with `cart = Cart(request)`. Depending on the request method, you 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, you create a new order in the database using `order = form.save()`. You iterate over the cart items and create an `OrderItem`

for each of them. Finally, you clear the cart's contents 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, as follows. The new line is highlighted in bold:

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

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

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

Add the `order_create` URL to the `href` HTML attribute, 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.

You still need to define templates for creating 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 add 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 method="post" class="order-form">
    {{ form.as_p }}
    <p><input type="submit" value="Place order"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

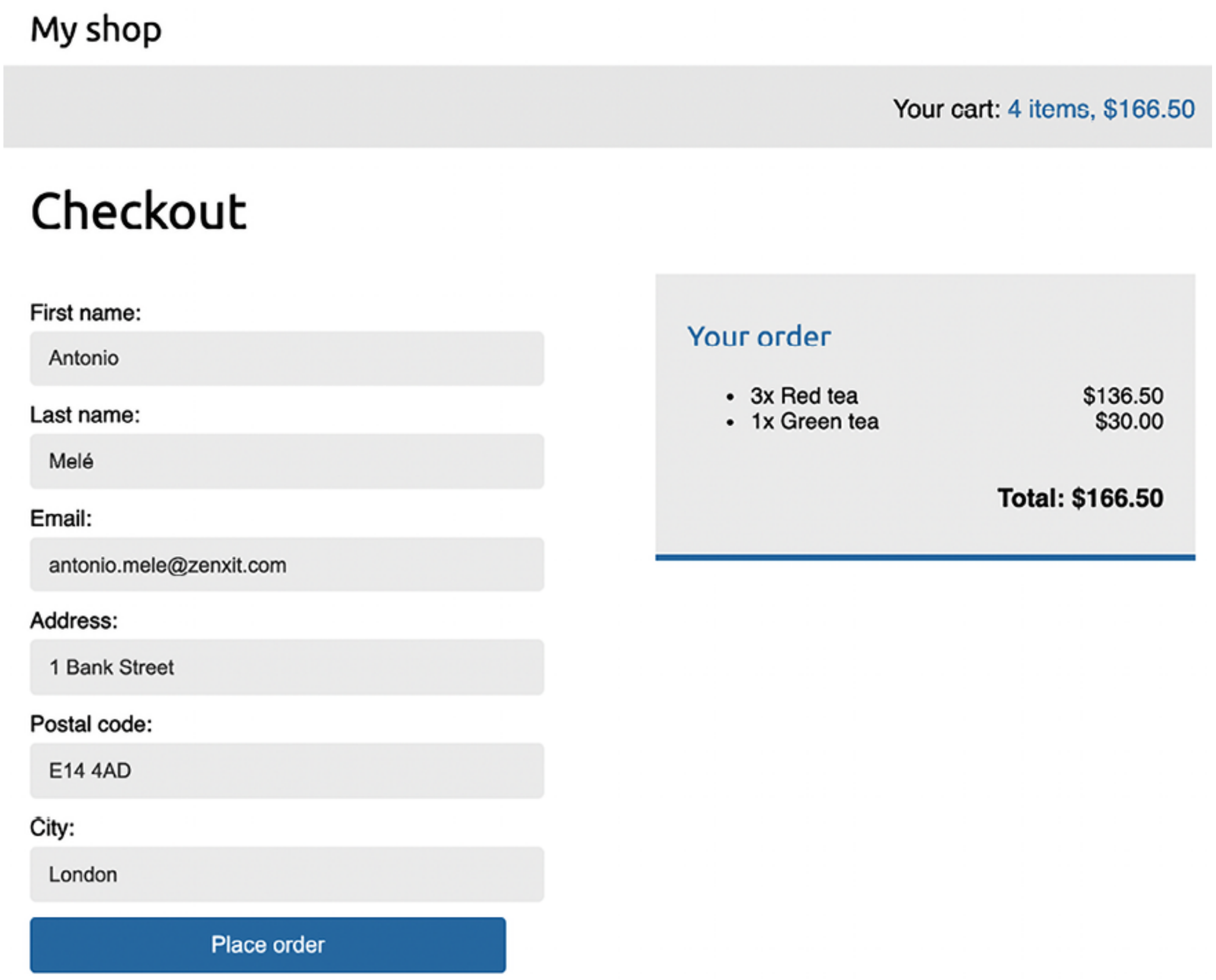
This template displays the cart items, including totals and the form to place an order.

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
  <strong>{{ order.id }}</strong>.</p>
{% endblock %}
```

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

Start the web development server to load 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 the following form:


My shop

Your cart: 4 items, $166.50

Checkout

First name: Antonio

Last name: Melé

Email: antonio.mele@zenxit.com

Address: 1 Bank Street

Postal code: E14 4AD

City: London

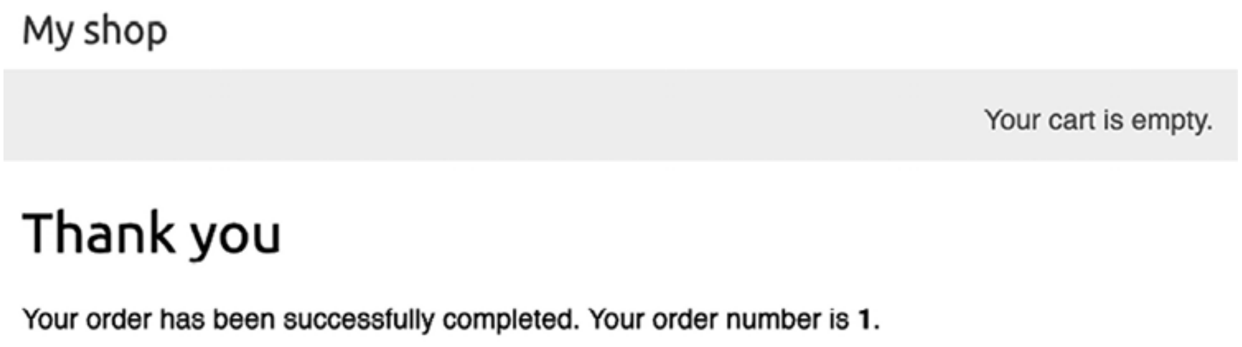
Your order

- 3x Red tea $136.50
- 1x Green tea $30.00

Total: $166.50

Figure 8.14: The order creation page, including the chart checkout form and order details

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


My shop

Your cart is empty.

Thank you

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

Figure 8.15: The order created template displaying the order number

The order has been registered and the cart has been cleared.

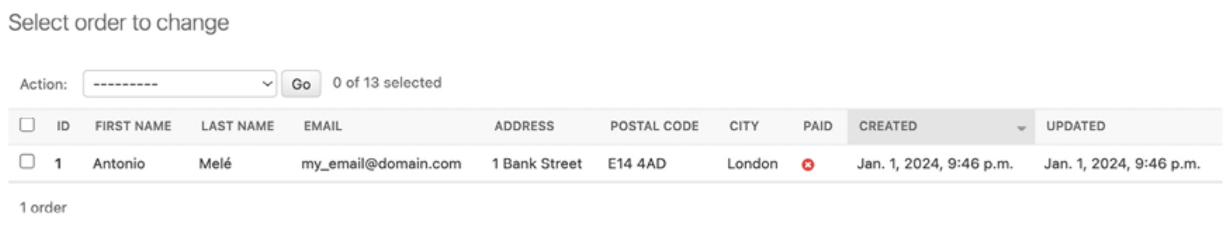
You might have noticed that the message **Your cart is empty** is displayed in the header when an order is completed. This is because the cart has been cleared. We can easily avoid this message for views that have an `order` object in the template context.

Edit the `shop/base.html` template of the `shop` application and replace the following line highlighted in bold:

```
...
<div class="cart">
  {% with total_items=cart|length %}
    {% if total_items > 0 %}
      Your cart:
      <a href="{% url "cart:cart_detail" %}">
        {{ total_items }} item{{ total_items|pluralize }},
        ${{ cart.get_total_price }}
      </a>
    {% elif not order %}
      Your cart is empty.
    {% endif %}
  {% endwith %}
</div>
...
```

The message **Your cart is empty** will not be displayed anymore when an order is created.

Now open the administration site at `http://127.0.0.1:8000/admin/orders/order/`. You will see that the order has been successfully created, like this:


Select order to change

Action: --------- Go 0 of 13 selected

ID	FIRST NAME	LAST NAME	EMAIL	ADDRESS	POSTAL CODE	CITY	PAID	CREATED	UPDATED
1	Antonio	Melé	my_email@domain.com	1 Bank Street	E14 4AD	London	✗	Jan. 1, 2024, 9:46 p.m.	Jan. 1, 2024, 9:46 p.m.

1 order

Figure 8.16: The order change list section of the administration site, including the order created

You have implemented the order system. Now you will learn how to create asynchronous tasks to send confirmation emails to users when they place an order.

Creating asynchronous tasks

When receiving an HTTP request, you need to return a response to the user as quickly as possible. Remember that in *Chapter 7, Tracking User Actions*, you used the Django Debug Toolbar to check the time for the different phases of the request/response cycle and the execution time for the SQL queries performed.

Every task executed during the course of the request/response cycle adds up to the total response time. Long-running tasks can seriously slow down the server response. How do we return a fast response to the user while still completing time-consuming tasks? We can do it with asynchronous execution.

Working with asynchronous tasks

We can offload work from the request/response cycle by executing certain tasks in the background. For example, a video-sharing platform allows users to upload videos but requires a long time to transcode uploaded videos. When the user uploads a video, the site might return a response informing them that the transcoding will start soon and start transcoding the video asynchronously. Another example is sending emails to users. If your site sends email notifications from a view, the **Simple Mail Transfer Protocol** (**SMTP**) connection might fail or slow down the response. By sending the email asynchronously, you avoid blocking the code execution.

Asynchronous execution is especially relevant for data-intensive, resource-intensive, and time-consuming processes or processes subject to failure, which might require a retry policy.

Workers, message queues, and message brokers

While your web server processes requests and returns responses, you need a second task-based server, named **worker**, to process the asynchronous tasks. One or multiple workers can be running and executing tasks in the background. These workers can access the database, process files, send emails, and so on. Workers can even queue future tasks, all while keeping the main web server free to process HTTP requests.

To tell the workers what tasks to execute, we need to send **messages**. We communicate with brokers by adding messages to a **message queue**, which is basically a **first in, first out** (**FIFO**) data structure. When a broker becomes available, it takes the first message from the queue and starts executing the corresponding task. When finished, the broker takes the next message from the queue and executes the corresponding task. Brokers become idle when the message queue is empty. When using multiple brokers, each broker takes the first available message in order when they become available. The queue ensures that each broker only gets one task at a time and that no task is processed by more than one worker.

Figure 8.17 shows how a message queue works:

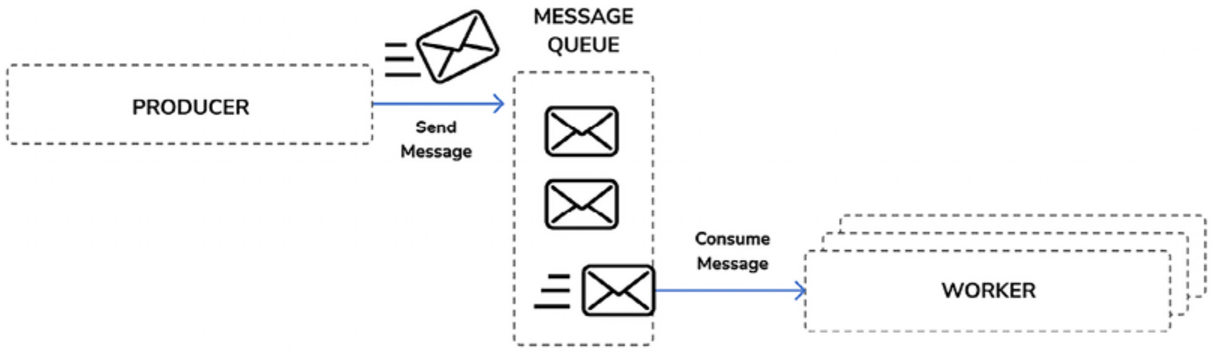


Figure 8.17: Asynchronous execution using a message queue and workers

A producer sends a message to the queue, and the worker(s) consumes the messages on a first-come, first-served basis; the first message added to the message queue is the first message to be processed by the worker(s).

In order to manage the message queue, we need a **message broker**. The message broker is used to translate messages to a formal messaging protocol and manage message queues for multiple receivers. It provides reliable storage and guaranteed message delivery. The message broker allows us to create message queues, route messages, distribute messages among workers, and so on.

To implement asynchronous tasks in your project, you will use Celery for managing task queues and RabbitMQ as the message broker Celery employs. Both technologies will be introduced in the following section.

Using Django with Celery and RabbitMQ

Celery is a distributed task queue that can process vast amounts of messages. We will use Celery to define asynchronous tasks as Python functions within our Django applications. We will run Celery workers that will listen to the message broker to get new messages to process asynchronous tasks.

Using Celery, not only can you create asynchronous tasks easily and let them be executed by workers as soon as possible but you can also schedule them to run at a specific time. You can find the Celery documentation at `https://docs.celeryq.dev/en/stable/index.html`.

Celery communicates via messages and requires a message broker to mediate between clients and workers. There are several options for a message broker for Celery, including key/value stores such as Redis, or an actual message broker such as RabbitMQ.

RabbitMQ is the most widely deployed message broker. It supports multiple messaging protocols, such as the **Advanced Message Queuing Protocol** (**AMQP**), and it is the recommended message worker for Celery. RabbitMQ is lightweight, easy to deploy, and can be configured for scalability and high availability.

Figure 8.18 shows how we will use Django, Celery, and RabbitMQ to execute asynchronous tasks:

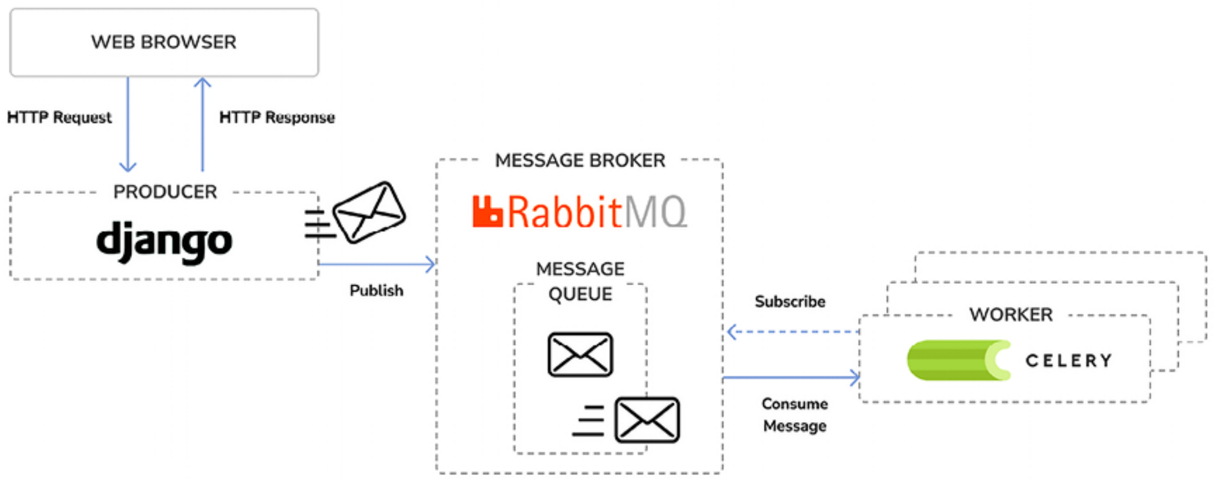


Figure 8.18: Architecture for asynchronous tasks with Django, RabbitMQ, and Celery

Installing Celery

Let's install Celery and integrate it into the project. Install Celery via `pip` using the following command:

```
python -m pip install celery==5.4.0
```

You can find an introduction to Celery at https://docs.celeryq.dev/en/stable/getting-started/introduction.html.

Installing RabbitMQ

The RabbitMQ community provides a Docker image that makes it very easy to deploy a RabbitMQ server with a standard configuration. Remember that you learned how to install Docker in *Chapter 3, Extending Your Blog Application*.

After installing Docker on your machine, you can easily pull the RabbitMQ Docker image by running the following command from the shell:

```
docker pull rabbitmq:3.13.1-management
```

This will download the RabbitMQ Docker image to your local machine. You can find information about the official RabbitMQ Docker image at https://hub.docker.com/_/rabbitmq.

If you want to install RabbitMQ natively on your machine instead of using Docker, you will find detailed installation guides for different operating systems at https://www.rabbitmq.com/download.html.

Execute the following command in the shell to start the RabbitMQ server with Docker:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672
```

With this command, we are telling RabbitMQ to run on port `5672`, and we are running its web-based management user interface on port `15672`.

You will see output that includes the following lines:

```
Starting broker...
...
completed with 4 plugins.
Server startup complete; 4 plugins started.
```

RabbitMQ is running on port `5672` and ready to receive messages.

Accessing RabbitMQ’s management interface

Open `http://127.0.0.1:15672/` in your browser. You will see the login screen for the management UI of RabbitMQ. It will look like this:

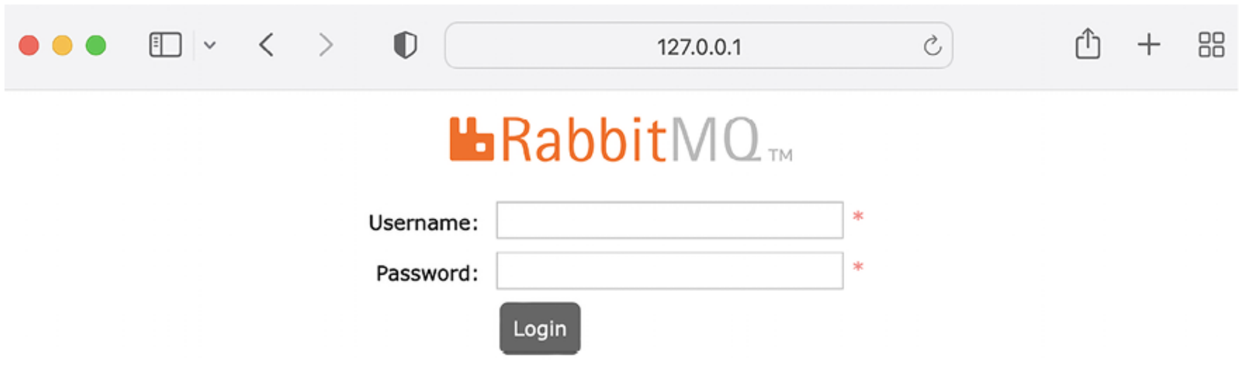
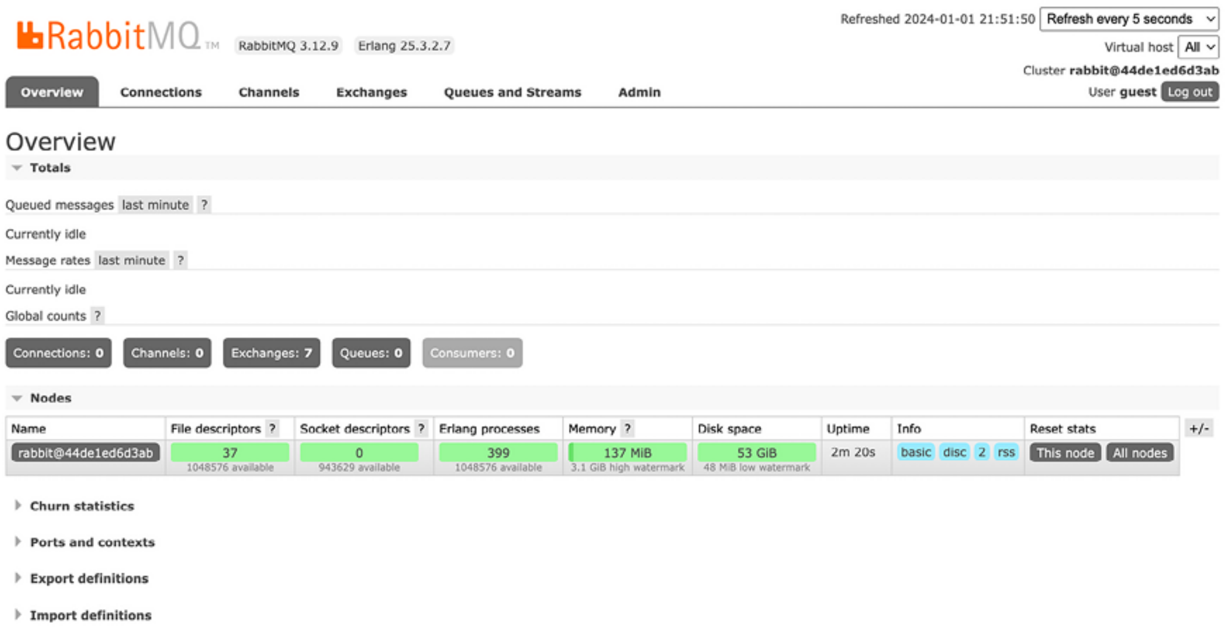


Figure 8.19: The RabbitMQ management UI login screen

Enter `guest` as both the username and the password and click on **Login**. You will see the following screen:



Refreshed 2024-01-01 21:51:50 Refresh every 5 seconds
Virtual host All
Cluster rabbit@44de1ed6d3ab
User guest

Overview

Totals

Queued messages last minute ?

Currently idle

Message rates last minute ?

Currently idle

Global counts ?

Connections: 0 Channels: 0 Exchanges: 7 Queues: 0 Consumers: 0

Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats
rabbit@44de1ed6d3ab	37 1048576 available	0 943629 available	399 1048576 available	137 MiB 3.1 GiB high watermark	53 GiB 48 MiB low watermark	2m 20s	basic disc 2 rss	This node All nodes

Figure 8.20: The RabbitMQ management UI dashboard

This is the default admin user for RabbitMQ. On this screen, you can monitor the current activity for RabbitMQ. You can see that there is one node running with no connections or queues registered.

If you use RabbitMQ in a production environment, you will need to create a new admin user and remove the default `guest` user. You can do that in the **Admin** section of the management UI.

Now we will add Celery to the project. Then, we will run Celery and test the connection to RabbitMQ.

Adding Celery to your project

You have to provide a configuration for the Celery instance. Create a new file next to the `settings.py` file of `myshop` and name it `celery.py`. This file will contain the Celery configuration for your project. Add the following code to it:

```
import os
from celery import Celery
# set the default Django settings module for the 'celery' progra
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myshop.settings
app = Celery('myshop')
app.config_from_object('django.conf:settings', namespace='CELERY
app.autodiscover_tasks()
```

In this code, you do the following:

- You set the `DJANGO_SETTINGS_MODULE` variable for the Celery command-line program.
- You create an instance of the application with `app = Celery('myshop')`.
- You load any custom configuration from your project settings using the `config_from_object()` method. The `namespace` attribute specifies the prefix that Celery-related settings will have in your `settings.py` file.

By setting the `CELERY` namespace, all Celery settings need to include the `CELERY_` prefix in their name (for example, `CELERY_BROKER_URL`).

- Finally, you tell Celery to auto-discover asynchronous tasks for your applications. Celery will look for a `tasks.py` file in each application directory of applications added to `INSTALLED_APPS` in order to load asynchronous tasks defined in it.

You need to import the `celery` module in the `__init__.py` file of your project to ensure it is loaded when Django starts.

Edit the `myshop/__init__.py` file and add the following code to it:

```
# import celery
from .celery import app as celery_app
__all__ = ['celery_app']
```

You have added Celery to the Django project, and you can now start using it.

Running a Celery worker

A Celery worker is a process that handles bookkeeping features like sending/receiving queue messages, registering tasks, killing hung tasks, tracking status, and so on. A worker instance can consume from any number of message queues.

Open another shell and start a Celery worker from your project directory, using the following command:

```
celery -A myshop worker -l info
```

The Celery worker is now running and ready to process tasks. Let’s check if there is a connection between Celery and RabbitMQ.

Open `http://127.0.0.1:15672/` in your browser to access the RabbitMQ management UI. You will now see a graph under **Queued messages** and another graph under **Message rates**, as in *Figure 8.21*:

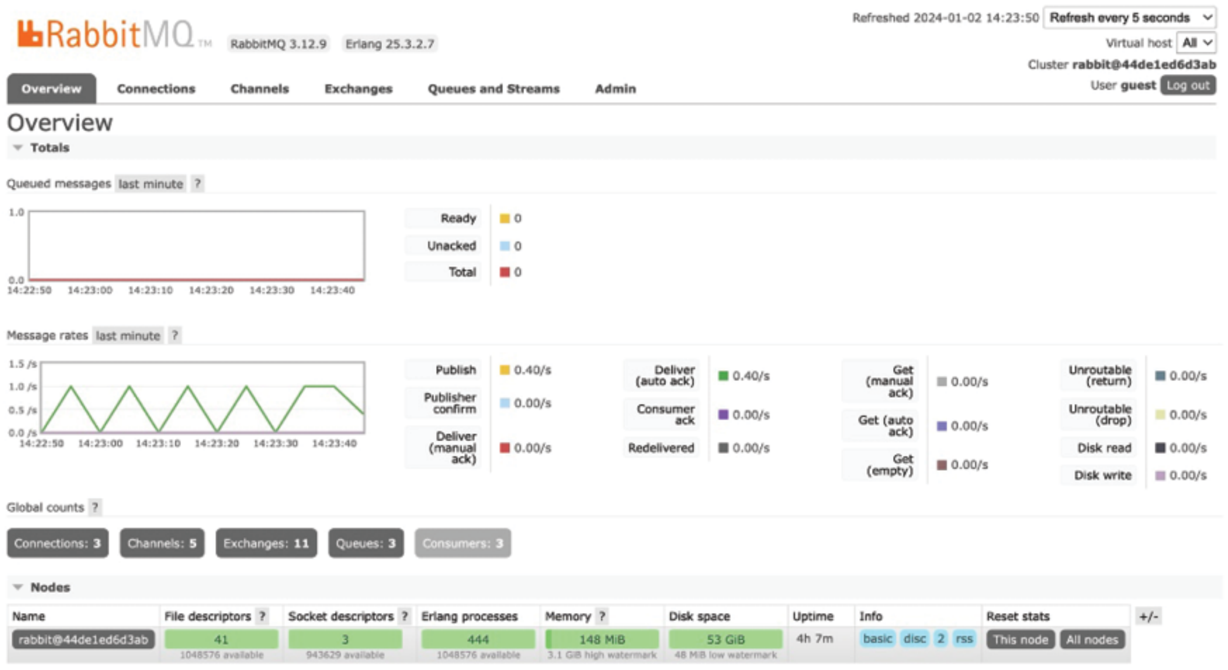


Figure 8.21: The RabbitMQ management dashboard displaying connections and queues

Obviously, there are no queued messages as we haven't sent any messages to the message queue yet. The graph under **Message rates** should update every five seconds; you can see the refresh rate at the top right of the screen. This time, both **Connections** and **Queues** should display a number higher than zero.

Now we can start programming asynchronous tasks.

The `CELERY_ALWAYS_EAGER` setting allows you to execute tasks locally in a synchronous manner instead of sending them to the queue. This is useful for running unit tests or executing the application in your local environment without running Celery.

Adding asynchronous tasks to your application

Let’s send a confirmation email to the user whenever an order is placed in the online shop. We will implement sending the email in a Python function and register it as a task with Celery. Then, we will add it to the `order_create` view to execute the task asynchronously.

When the `order_create` view is executed, Celery will send the message to a message queue managed by RabbitMQ and then a Celery broker will execute the asynchronous task that we defined with a Python function.

The convention for easy task discovery by Celery is to define asynchronous tasks for your application in a `tasks` module within the application directory.

Create a new file inside the `orders` application and name it `tasks.py`. This is the place where Celery will look for asynchronous tasks. Add the following code to it:

```
from celery import shared_task
from django.core.mail import send_mail
from .models import Order
@shared_task
def order_created(order_id):
    """
    Task to send an e-mail notification when an order is
    successfully created.
    """
    order = Order.objects.get(id=order_id)
    subject = f'Order nr. {order.id}'
    message = (
        f'Dear {order.first_name},\n\n'
f'You have successfully placed an order.'
f'Your order ID is {order.id}.'
    )
    mail_sent = send_mail(
        subject, message, 'admin@myshop.com', [order.email]
    )
    return mail_sent
```

We have defined the `order_created` task by using the `@shared_task` decorator. As you can see, a Celery task is just a Python function decorated with `@shared_task`. The `order_created` task function receives an `order_id` parameter. It’s always recommended to only pass IDs to task functions and retrieve objects from the database when the task is executed. By doing so, we avoid accessing outdated information since the data in the database might have changed while the task was queued. We have used the `send_mail()` function provided by Django to send an email notification to the user who placed the order.

You learned how to configure Django to use your SMTP server in *Chapter 2, Enhancing Your Blog with Advanced Features*. If you don’t want to set up email settings, you can tell Django to write emails to the console by adding the following setting to the `settings.py` file:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Use asynchronous tasks not only for time-consuming processes but also for other processes that do not take so much time to be executed but that are subject to connection failures or require a retry policy.

Now you have to add the task to your `order_create` view. Edit the `views.py` file of the `orders` application, import the task, and call the `order_created` asynchronous task after clearing the cart, as follows:

```
# ...
from .tasks import order_created
def order_create(request):
    # ...
if request.method == 'POST':
        # ...
if form.is_valid():
            # ...
            cart.clear()
 # launch asynchronous task
            order_created.delay(order.id)
# ...
```

You call the `delay()` method of the task to execute it asynchronously. The task will be added to the message queue and executed by the Celery worker as soon as possible.

Make sure RabbitMQ is running. Then, stop the Celery worker process and start it again with the following command:

```
celery -A myshop worker -l info
```

The Celery worker has now registered the task. In another shell, start the development server from the project directory with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/` in your browser, add some products to your shopping cart, and complete an order. In the shell where you started the

Celery worker, you will see output similar to the following:

```
[2024-01-02 20:25:19,569: INFO/MainProcess] Task orders.tasks.or
...
[2024-01-02 20:25:19,605: INFO/ForkPoolWorker-8] Task orders.tas
```

The `order_created` task has been executed and an email notification for the order has been sent. If you are using the email backend `console.EmailBackend`, no email is sent but you should see the rendered text of the email in the output of the console.

Monitoring Celery with Flower

Besides the RabbitMQ management UI, you can use other tools to monitor the asynchronous tasks that are executed with Celery. Flower is a useful web-based tool for monitoring Celery. You can find the documentation for Flower at https://flower.readthedocs.io/.

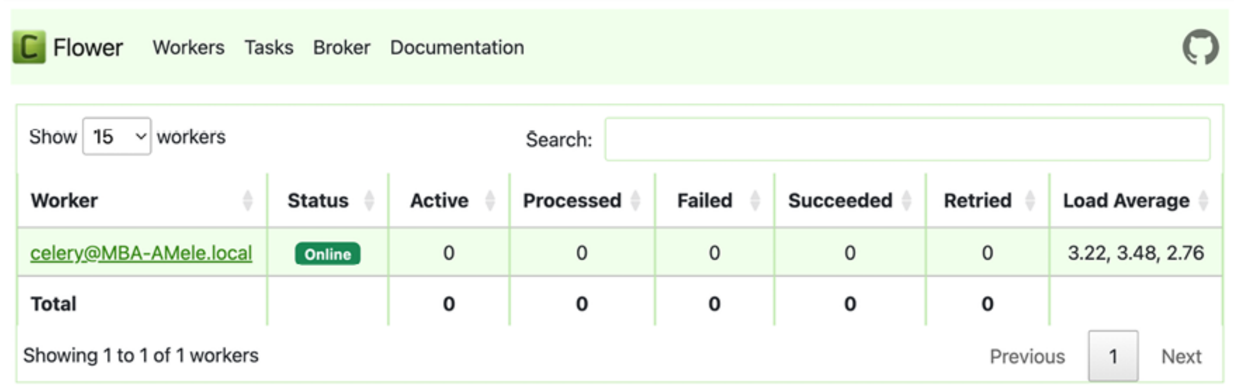
Install Flower using the following command:

```
python -m pip install flower==2.0.1
```

Once installed, you can launch Flower by running the following command in a new shell from your project directory:

```
celery -A myshop flower
```

Open `http://localhost:5555/` in your browser. You will be able to see the active Celery workers and asynchronous task statistics. The screen should look as follows:



Show 15 workers Search:

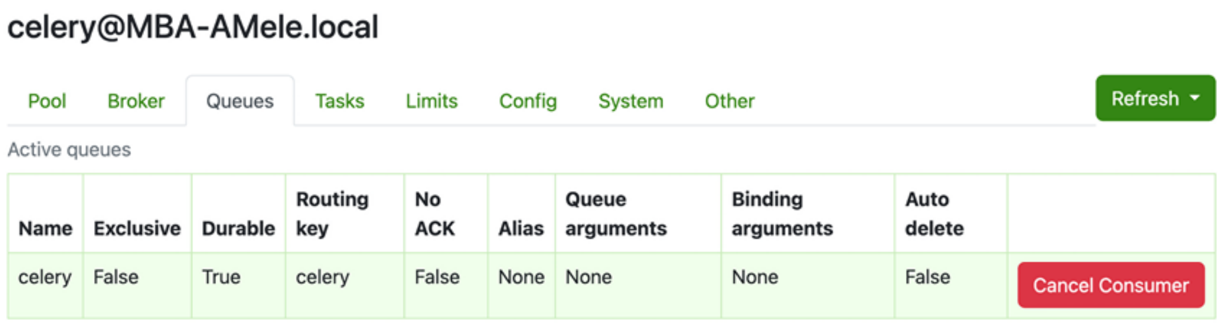
Worker	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@MBA-AMele.local	Online	0	0	0	0	0	3.22, 3.48, 2.76
Total		**0**	**0**	**0**	**0**	**0**	

Showing 1 to 1 of 1 workers

Figure 8.22: The Flower dashboard

You will see an active worker whose name starts with **celery@** and whose status is **Online**.

Click on the worker's name and then click on the **Queues** tab. You will see the following screen:


celery@MBA-AMele.local

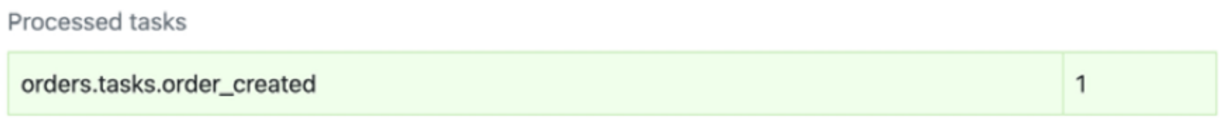
Active queues

Name	Exclusive	Durable	Routing key	No ACK	Alias	Queue arguments	Binding arguments	Auto delete	
celery	False	True	celery	False	None	None	None	False	Cancel Consumer

Figure 8.23: Flower – Worker Celery task queues

Here you can see the active queue named **celery**. This is the active queue consumer connected to the message broker.

Click the **Tasks** tab. You will see the following screen:


Processed tasks

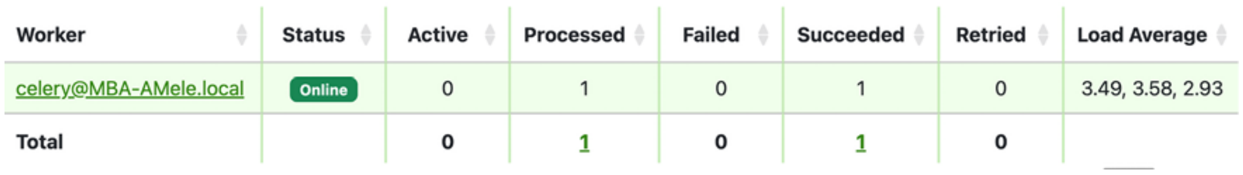
orders.tasks.order_created	1

Figure 8.24: Flower – Worker Celery tasks

Here you can see the tasks that have been processed and the number of times that they have been executed. You should see the `order_created` task and the total times that it has been executed. This number might vary depending on how many orders you have placed.

Open `http://localhost:8000/` in your browser. Add some items to the cart and then complete the checkout process.

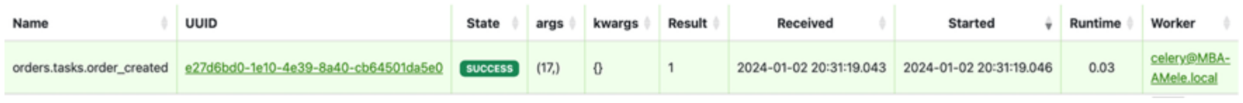
Open `http://localhost:5555/` in your browser. Flower has registered the task as processed. You should now see `1` under **Processed** and `1` under **Succeeded** as well:



Worker	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@MBA-AMele.local	Online	0	1	0	1	0	3.49, 3.58, 2.93
Total		0	1	0	1	0	

Figure 8.25: Flower – Celery workers

Under **Tasks**, you can see additional details about each task registered with Celery:



Name	UUID	State	args	kwargs	Result	Received	Started	Runtime	Worker
orders.tasks.order_created	e27d6bd0-1e10-4e39-8a40-cb64501da5e0	SUCCESS	(17,)	{}	1	2024-01-02 20:31:19.043	2024-01-02 20:31:19.046	0.03	celery@MBA-AMele.local

Figure 8.26: Flower – Celery tasks

Flower should never be deployed openly in a production environment without security. Let's add authentication to the Flower instance. Stop Flower using *Ctrl + C*, and restart it with the `--basic-auth` option by executing the following command:

```
celery -A myshop flower --basic-auth=user:pwd
```

Replace `user` and `pwd` with your desired username and password. Open `http://localhost:5555/` in your browser. The browser will now prompt you for credentials, as shown in *Figure 8.27*:

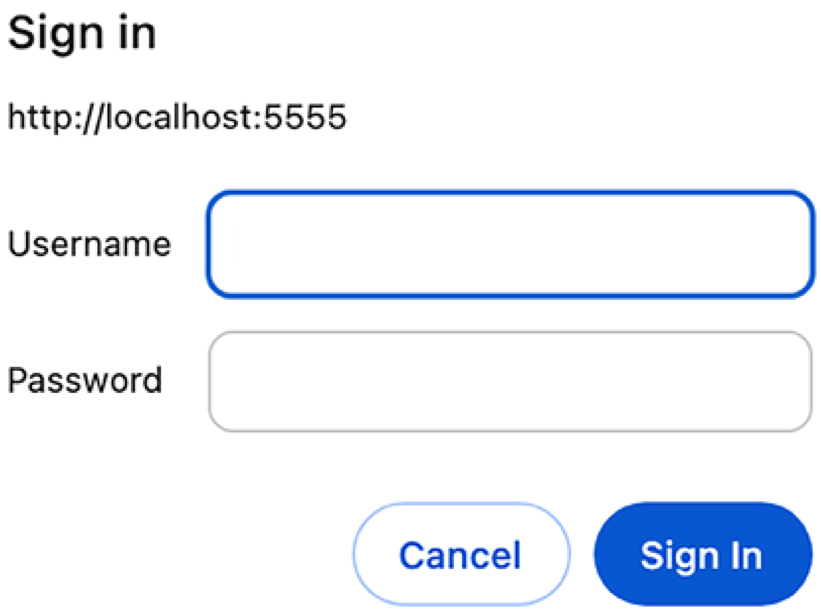


Figure 8.27: Basic authentication required to access Flower

Flower provides other authentication options, such as Google, GitHub, or Okta OAuth. You can read more about Flower's authentication methods at `https://flower.readthedocs.io/en/latest/auth.html`.

Summary

In this chapter, you created a basic e-commerce application. You made a product catalog and built a shopping cart using sessions. You implemented a custom context processor to make the cart available to all templates and created a form for placing orders. You also learned how to implement asynchronous tasks using Celery and RabbitMQ. Having completed this chapter, you now understand the foundational elements of building an e-commerce platform with Django, including managing products, processing orders, and handling asynchronous tasks. You are now also capable of developing projects that efficiently process user transactions and scale to handle complex background operations seamlessly.

In the next chapter, you will discover how to integrate a payment gateway into your shop, add custom actions to the administration site, export data in CSV format, and generate PDF files dynamically.

Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter – `https://github.com/PacktPublishing/Django-5-by-example/tree/main/Chapter08`
- Static files for the project – `https://github.com/PacktPublishing/Django-5-by-Example/tree/main/Chapter08/myshop/shop/static`
- Django session settings – `https://docs.djangoproject.com/en/5.0/ref/settings/#sessions`
- Django built-in context processors – `https://docs.djangoproject.com/en/5.0/ref/templates/api/#built-in-template-context-processors`

- Information about `RequestContext` – https://docs.djangoproject.com/en/5.0/ref/templates/api/#django.template.RequestContext
- Celery documentation – https://docs.celeryq.dev/en/stable/index.html
- Introduction to Celery – https://docs.celeryq.dev/en/stable/getting-started/introduction.html
- Official RabbitMQ Docker image – https://hub.docker.com/_/rabbitmq
- RabbitMQ installation instructions – https://www.rabbitmq.com/download.html
- Flower documentation – https://flower.readthedocs.io/
- Flower authentication methods – https://flower.readthedocs.io/en/latest/auth.html

Join us on Discord!

Read this book alongside other users, Django development experts, and the author himself. Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.Scan the QR code or visit the link to join the community.

https://packt.link/Django5ByExample