```
    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 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 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" %}{% endif %}">
              </a>
            </td>
```

```
            <td>{{ product.name }}</td>
            <td>{{ item.quantity }}</td>
            <td>
              <form action="{% url "cart:cart_remove" product.id %}"
method="post">
                <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 into 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_form})
```

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:

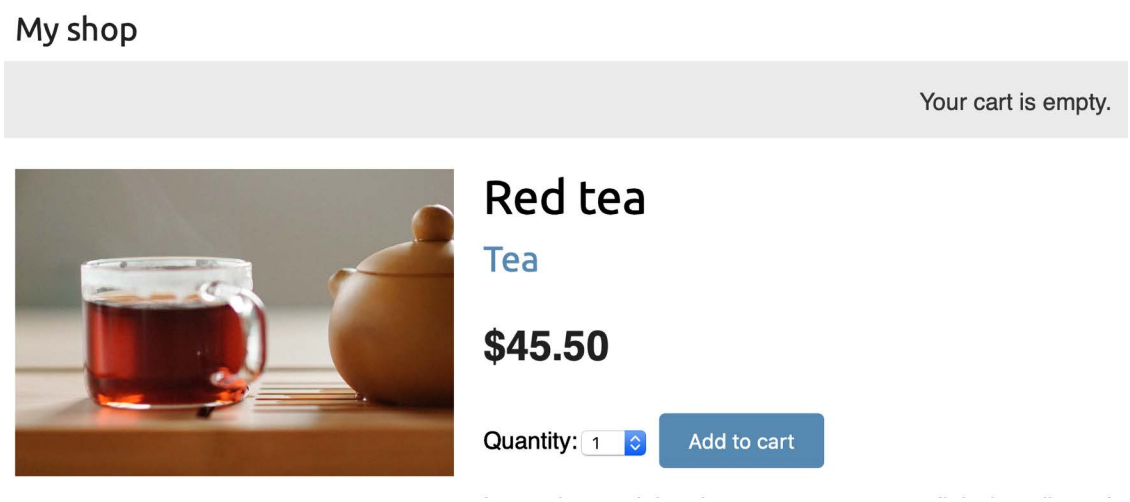


Figure 8.8: The product detail page, including the Add to cart form

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 8.9*:

Figure 8.9: 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': True})
    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="post">
```

```
    {{ 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:

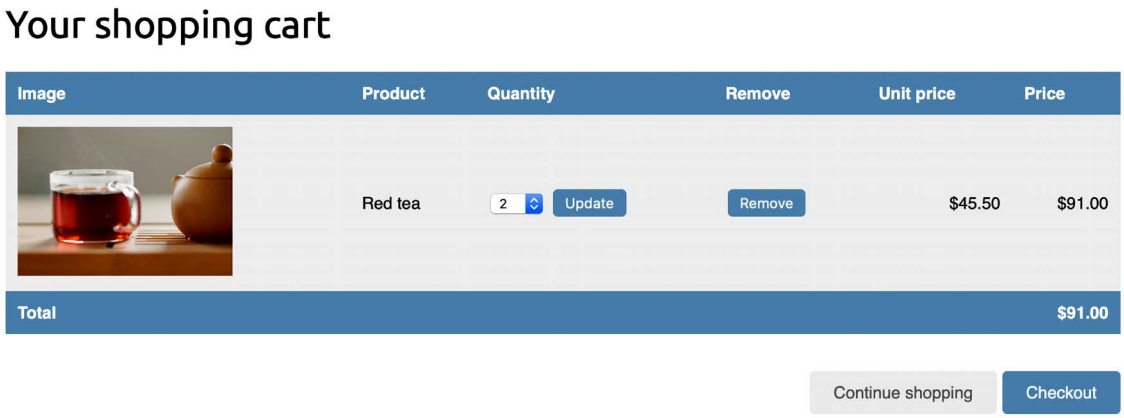


Figure 8.10: The cart detail page, including the form to update product quantities

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/4.1/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. 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.DjangoTemplates',
        '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.messages',
                '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/4.1/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:

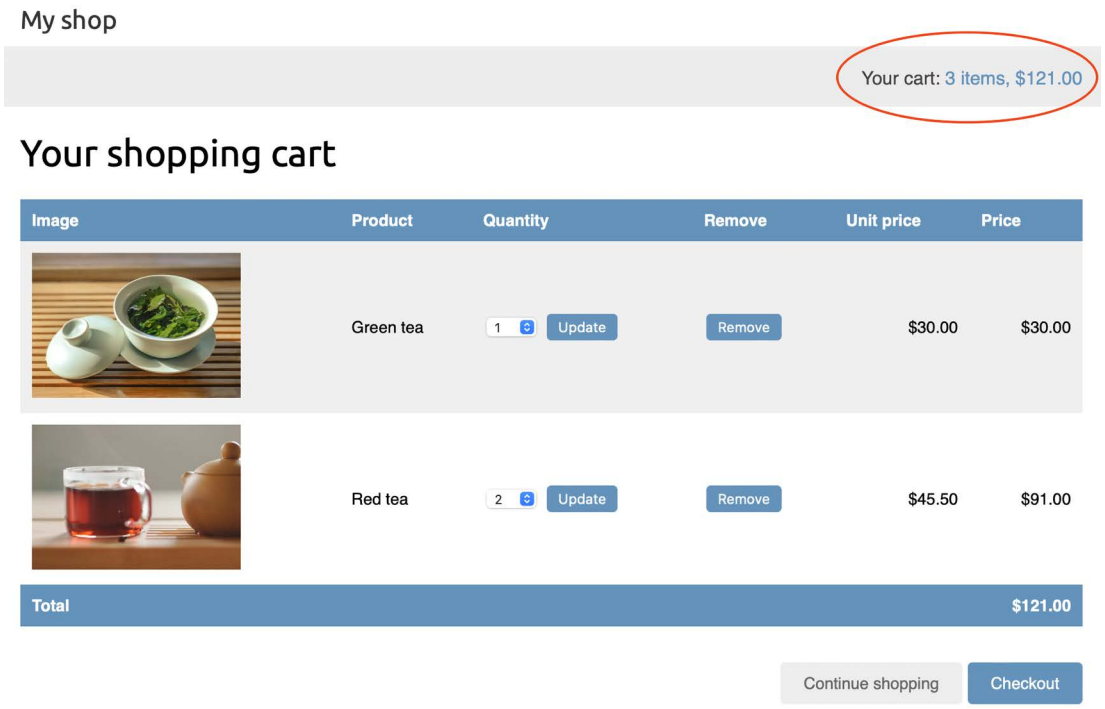


Figure 8.11: The site header displaying the current items in the cart

You have completed the cart functionality. Next, you are going to create the functionality to register customer orders.

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

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
    - Create index orders_orde_created_743fca_idx on field(s) -created of model
order
```

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:

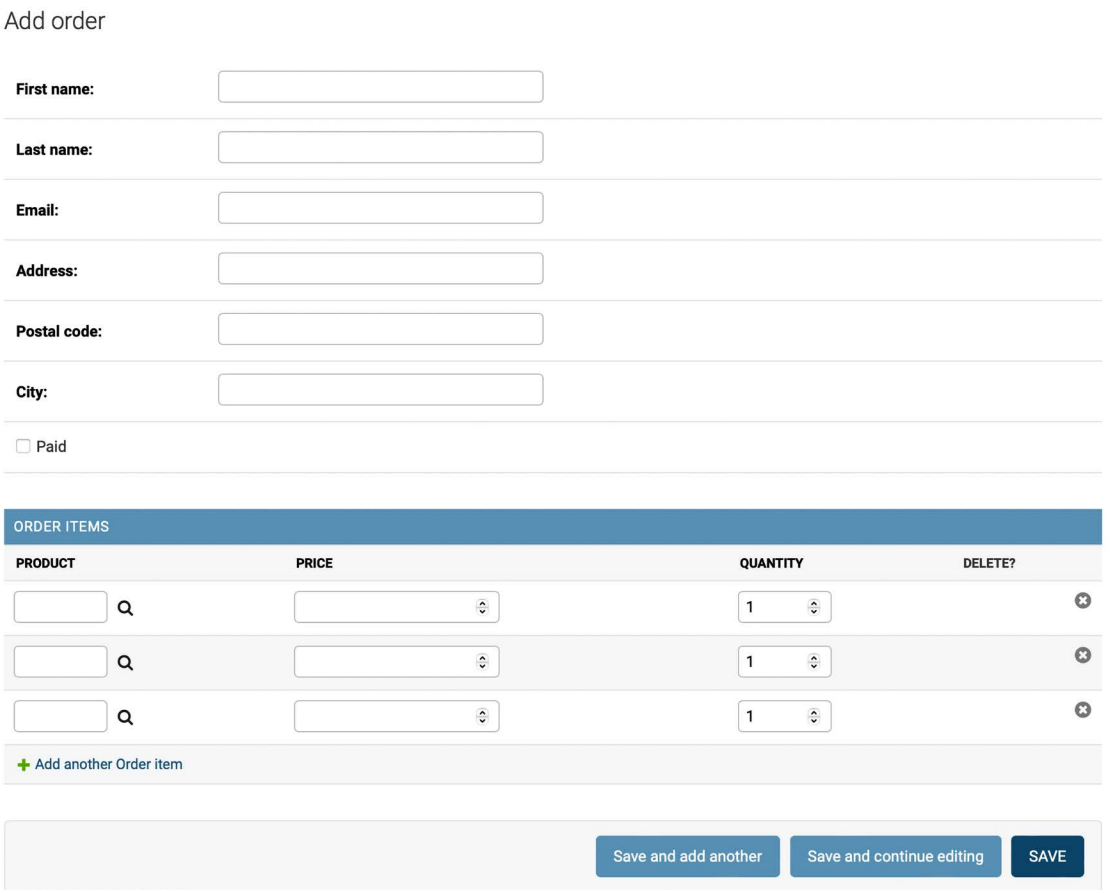


Figure 8.12: The Add order form, including OrderItemInline

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 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, 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 is
  <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:

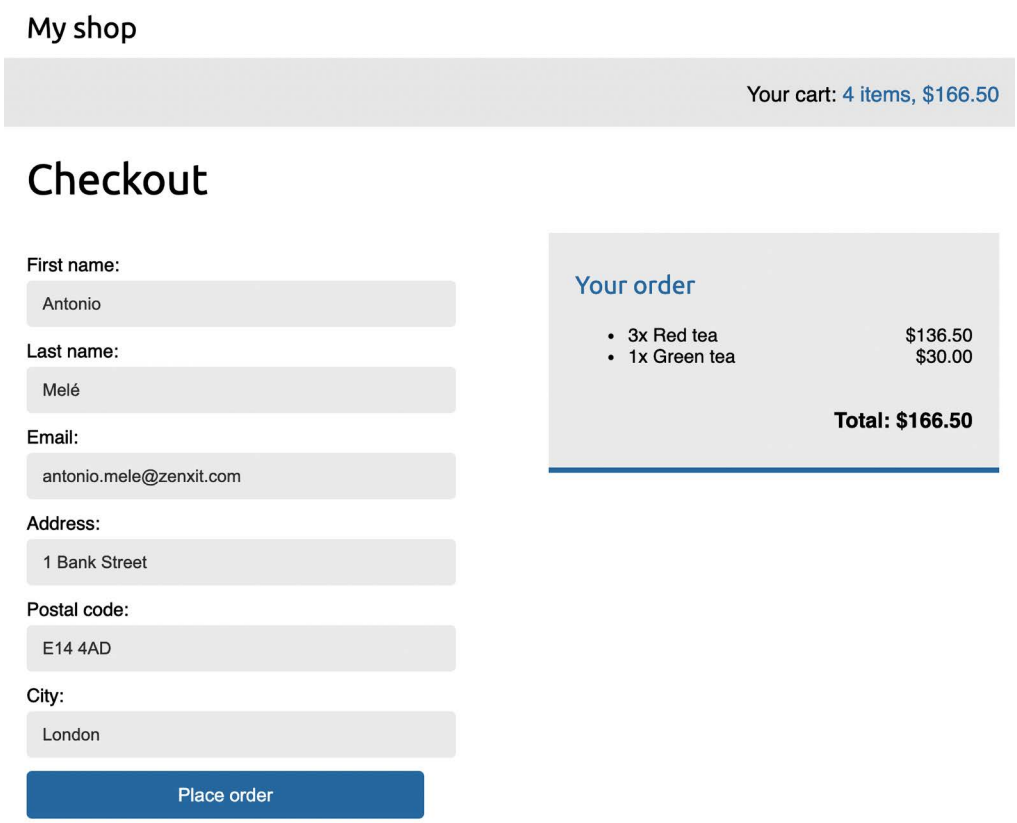


Figure 8.13: The order creation page, including the chart checkout form and order details

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:

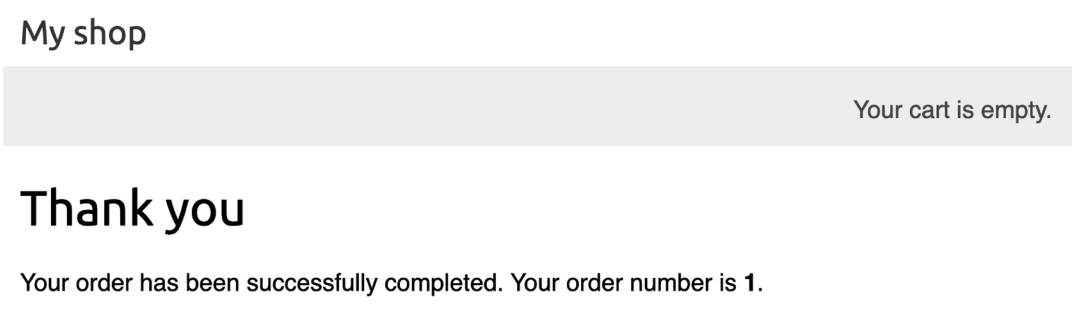


Figure 8.14: The order created template displaying the order number

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:

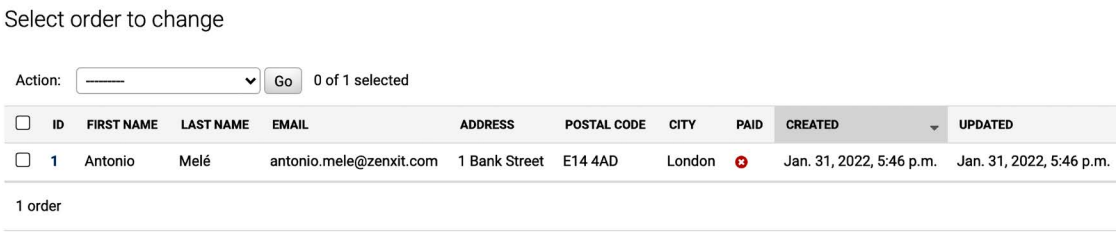


Figure 8.15: The order change list section of the administration site including the order created

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.

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 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 e-mails, etc. 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 each broker only gets one task at a time, and that no task is processed by more than one worker.

Figure 8.16 shows how a message queue works:

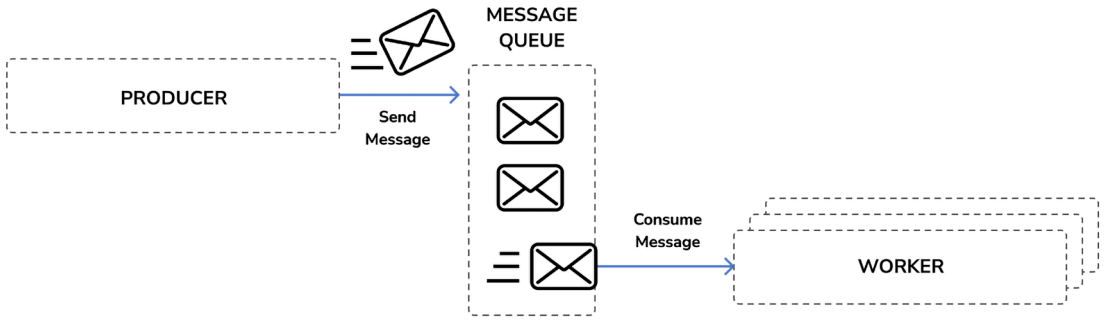


Figure 8.16: Asynchronous execution using a message queue and workers

A producer sends a message to the queue, and the worker(s) consume 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, etc.

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

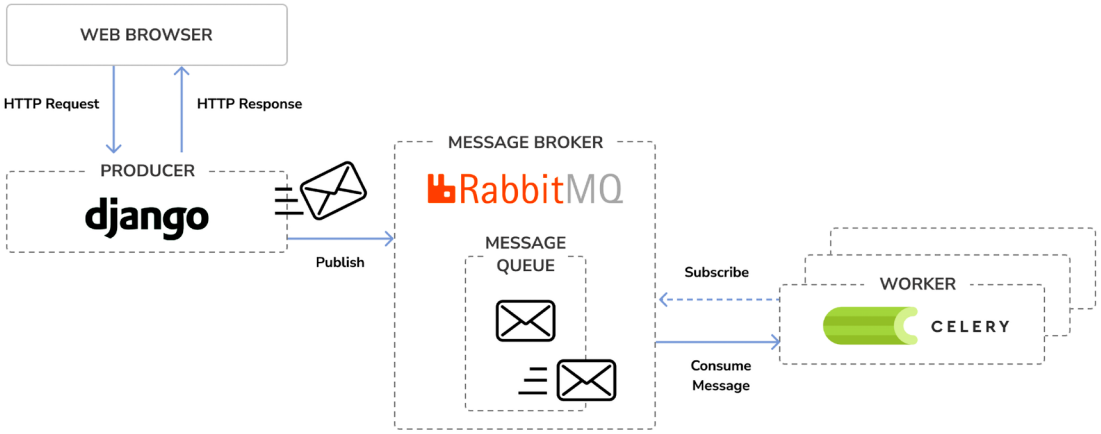


Figure 8.17: Architecture for asynchronous tasks with Django, RabbitMQ, and Celery

Installing Celery

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

```
pip install celery==5.2.7
```

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 7*, *Tracking User Actions*.

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
```

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
rabbitmq:management
```

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:

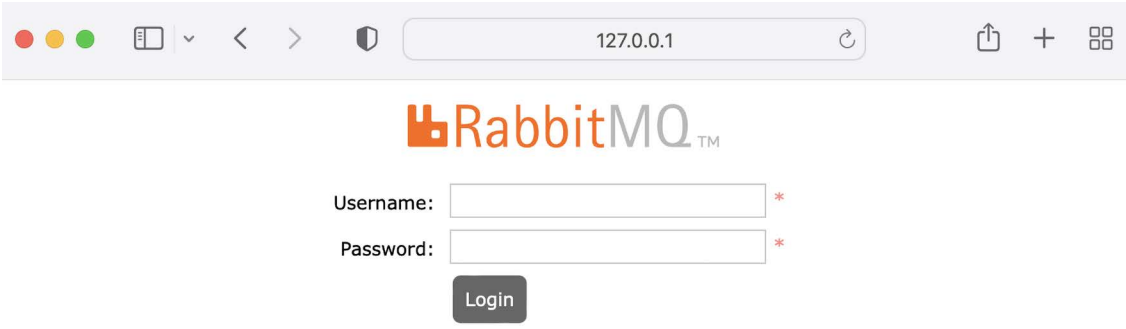


Figure 8.18: The RabbitMQ management UI login screen

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

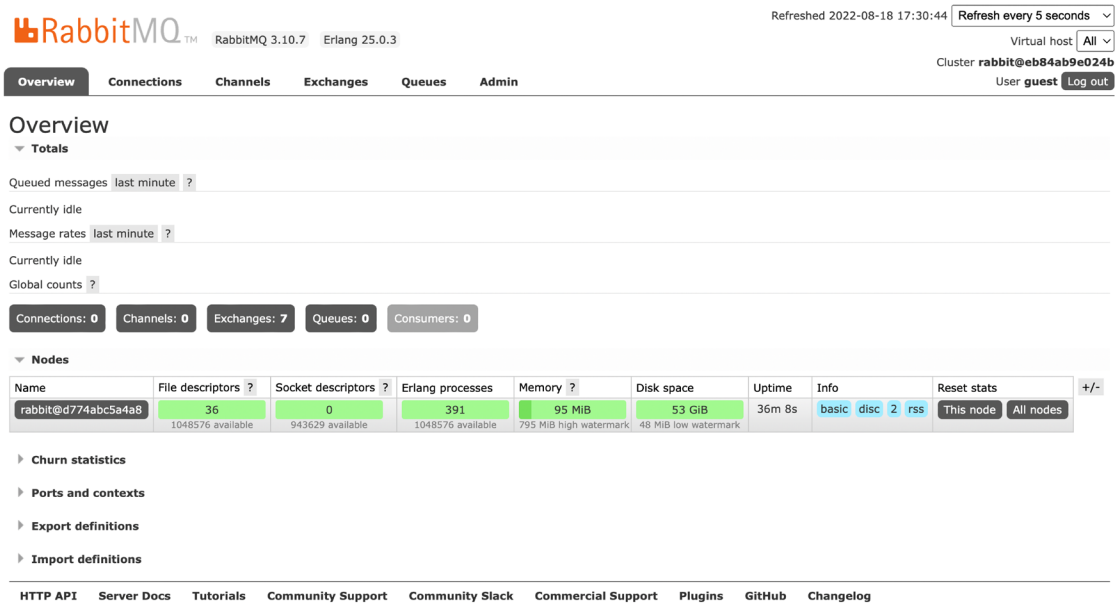


Figure 8.19: The RabbitMQ management UI dashboard

This is the default admin user for RabbitMQ. In 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' program.
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, etc. 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**, like in *Figure 8.20*:

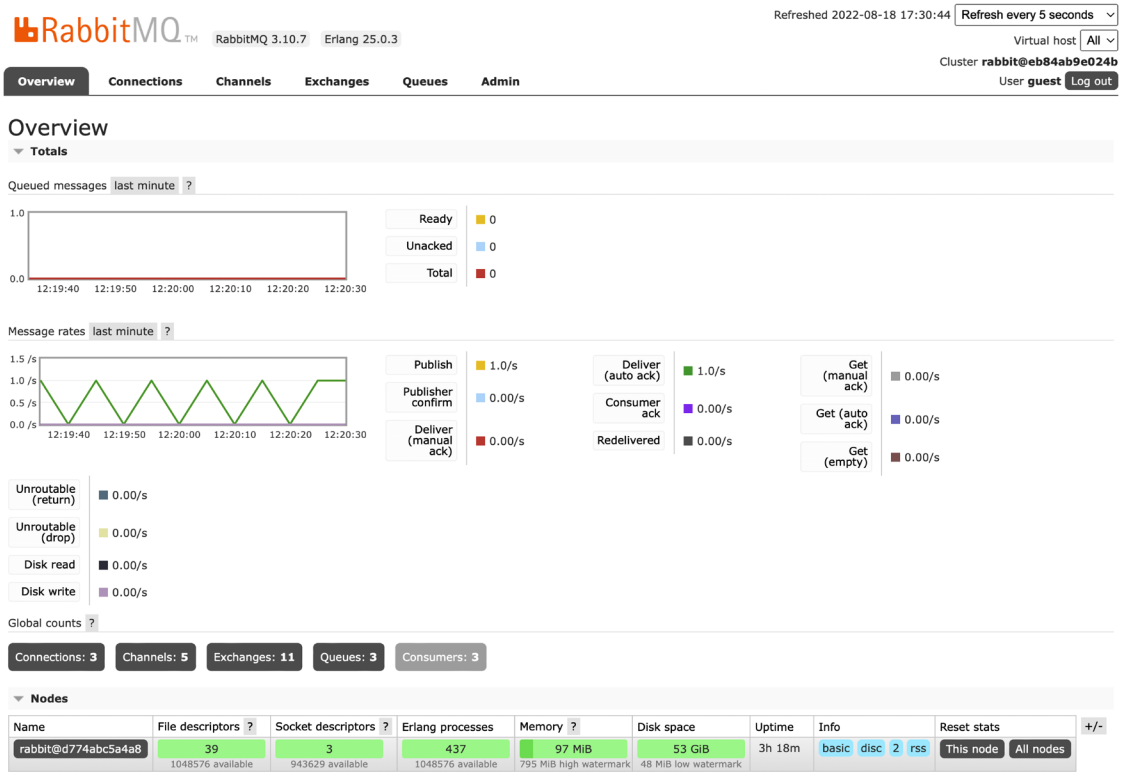


Figure 8.20: The RabbitMQ management dashboard displaying connections and queues

Obviously, there are no queued messages as we didn't send any messages to the message queue yet. The graph under **Message rates** should update every five seconds; you can see the refresh rate on 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:

```
[2022-02-03 20:25:19,569: INFO/MainProcess] Task orders.tasks.order_
created[a94dc22e-372b-4339-bff7-52bc83161c5c] received
...
[2022-02-03 20:25:19,605: INFO/ForkPoolWorker-8] Task orders.tasks.
order_created[a94dc22e-372b-4339-bff7-52bc83161c5c] succeeded in
0.015824042027816176s: 1
```

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.

Install Flower using the following command:

```
pip install flower==1.1.0
```

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

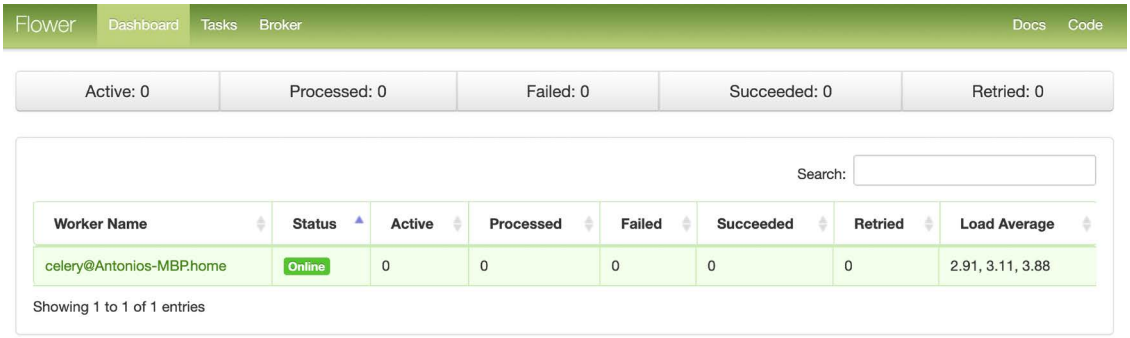


Figure 8.21: The Flower dashboard

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:

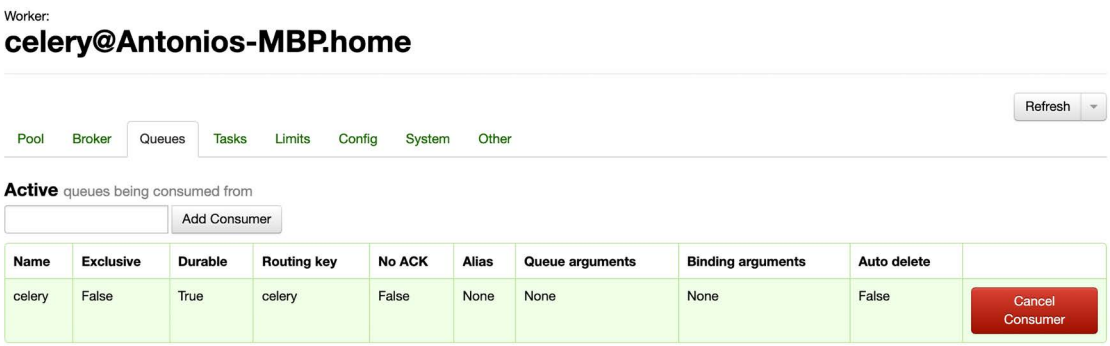


Figure 8.22: Flower – Worker Celery task queues

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:

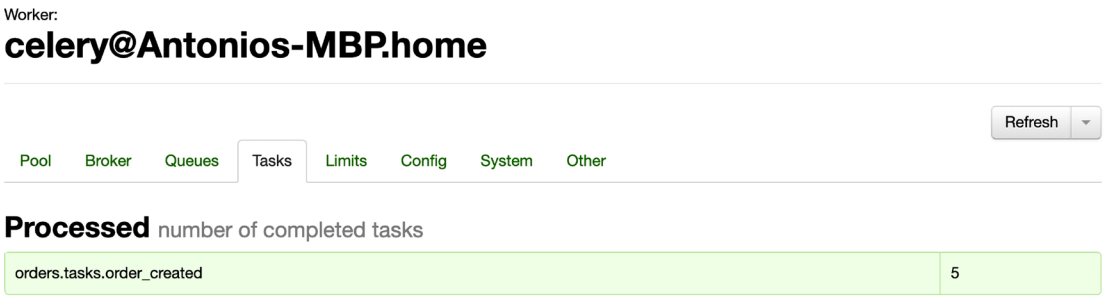


Figure 8.23: Flower – Worker Celery tasks

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

Figure 8.24: Flower – Celery workers

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

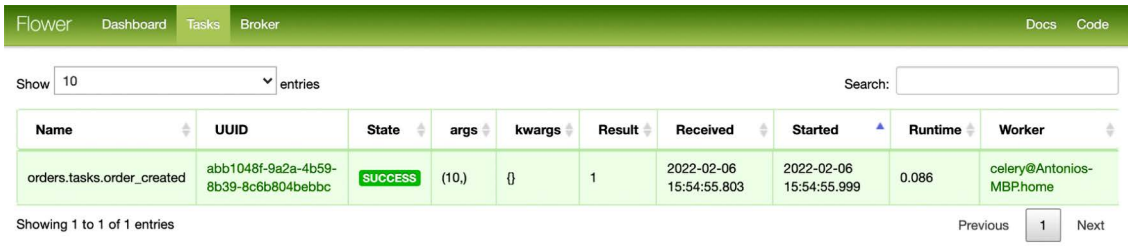


Figure 8.25: Flower – Celery tasks

You can find the documentation for Flower at `https://flower.readthedocs.io/`.

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-4-by-example/tree/main/Chapter08`
- Static files for the project – `https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter08/myshop/shop/static`
- Django session settings – `https://docs.djangoproject.com/en/4.1/ref/settings/#sessions`
- Django built-in context processors – `https://docs.djangoproject.com/en/4.1/ref/templates/api/#built-in-template-context-processors`
- Information about `RequestContext` – `https://docs.djangoproject.com/en/4.1/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/`

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.

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.

Join us on Discord

Read this book alongside other users and the author.

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 book community.

`https://packt.link/django`

9

Managing Payments and Orders

In the previous chapter, you created a basic online shop with a product catalog and a shopping cart. You learned how to use Django sessions and built a custom context processor. You also learned how to launch asynchronous tasks using Celery and RabbitMQ.

In this chapter, you will learn how to integrate a payment gateway into your site to let users pay by credit card. You will also extend the administration site with different features.

In this chapter, you will:

- Integrate the Stripe payment gateway into your project
- Process credit card payments with Stripe
- Handle payment notifications
- Export orders to CSV files
- Create custom views for the administration site
- Generate PDF invoices dynamically

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter09`.

All Python packages used in this chapter are included in the `requirements.txt` file in the source code for the chapter. You can follow the instructions to install each Python package in the following sections, or you can install all the requirements at once with the command `pip install -r requirements.txt`.

Integrating a payment gateway

A payment gateway is a technology used by merchants to process payments from customers online. Using a payment gateway, you can manage customers' orders and delegate payment processing to a reliable, secure third party. By using a trusted payment gateway, you won't have to worry about the technical, security, and regulatory complexity of processing credit cards in your own system.

There are several payment gateway providers to choose from. We are going to integrate Stripe, which is a very popular payment gateway used by online services such as Shopify, Uber, Twitch, and GitHub, among others.

Stripe provides an **Application Programming Interface** (**API**) that allows you to process online payments with multiple payment methods, such as credit card, Google Pay, and Apple Pay. You can learn more about Stripe at `https://www.stripe.com/`.

Stripe provides different products related to payment processing. It can manage one-off payments, recurring payments for subscription services, multiparty payments for platforms and marketplaces, and more.

Stripe offers different integration methods, from Stripe-hosted payment forms to fully customizable checkout flows. We will integrate the *Stripe Checkout* product, which consists of a payment page optimized for conversion. Users will be able to easily pay with a credit card or other payment methods for the items they order. We will receive payment notifications from Stripe. You can see the *Stripe Checkout* documentation at `https://stripe.com/docs/payments/checkout`.

By leveraging *Stripe Checkout* to process payments, you rely on a solution that is secure and compliant with **Payment Card Industry** (**PCI**) requirements. You will be able to collect payments from Google Pay, Apple Pay, Afterpay, Alipay, SEPA direct debits, Bacs direct debit, BECS direct debit, iDEAL, Sofort, GrabPay, FPX, and other payment methods.

Creating a Stripe account

You need a Stripe account to integrate the payment gateway into your site. Let's create an account to test the Stripe API. Open `https://dashboard.stripe.com/register` in your browser. You will see a form like the following one:

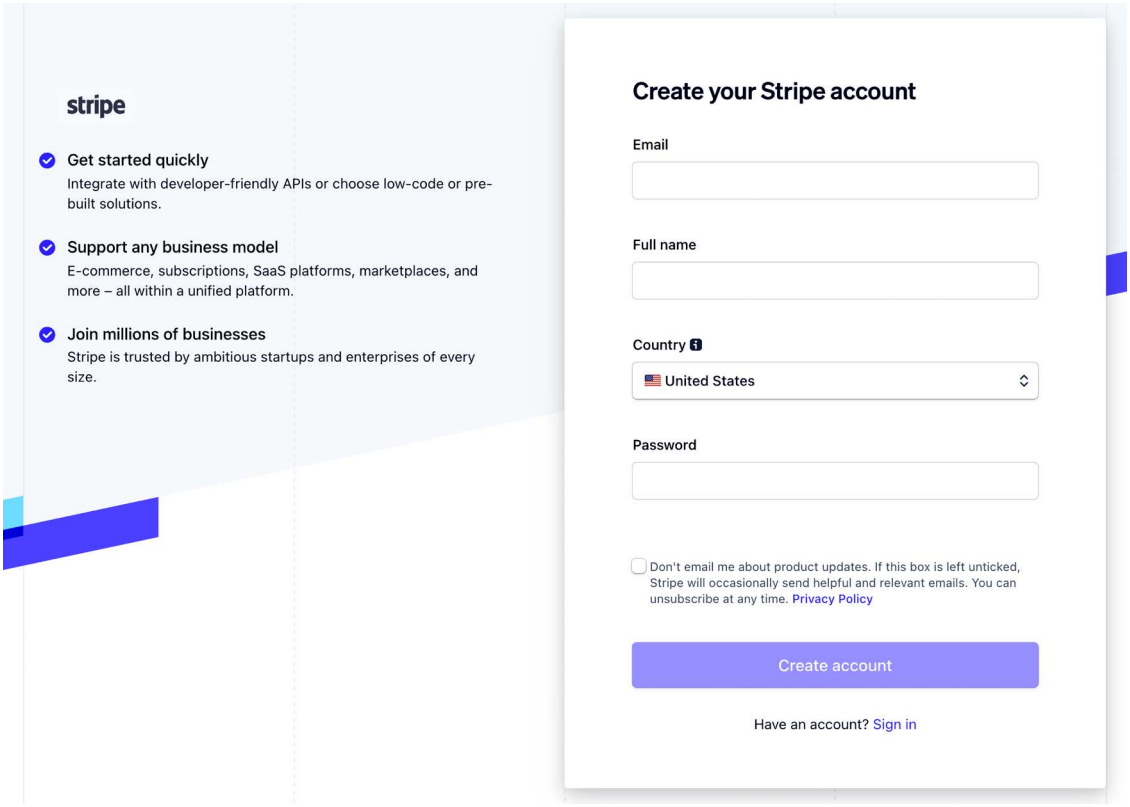


Figure 9.1: The Stripe signup form

Fill in the form with your own data and click on **Create account**. You will receive an email from Stripe with a link to verify your email address. The email will look like this:

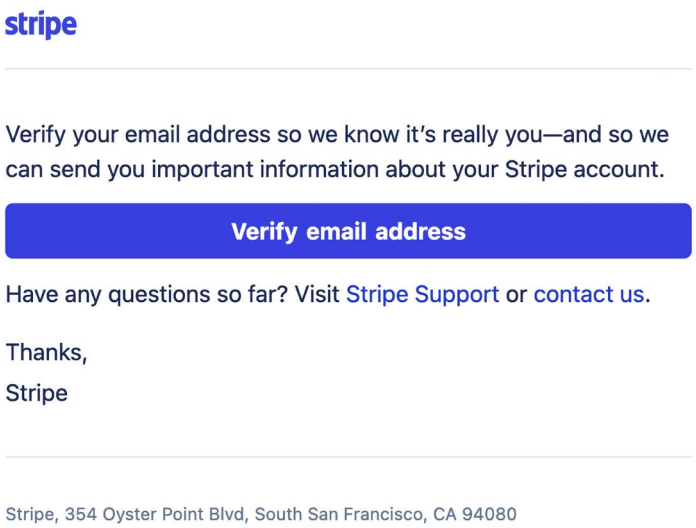


Figure 9.2: The verification email to verify your email address

Open the email in your inbox and click on **Verify email address**.

You will be redirected to the Stripe dashboard screen, which will look like this:

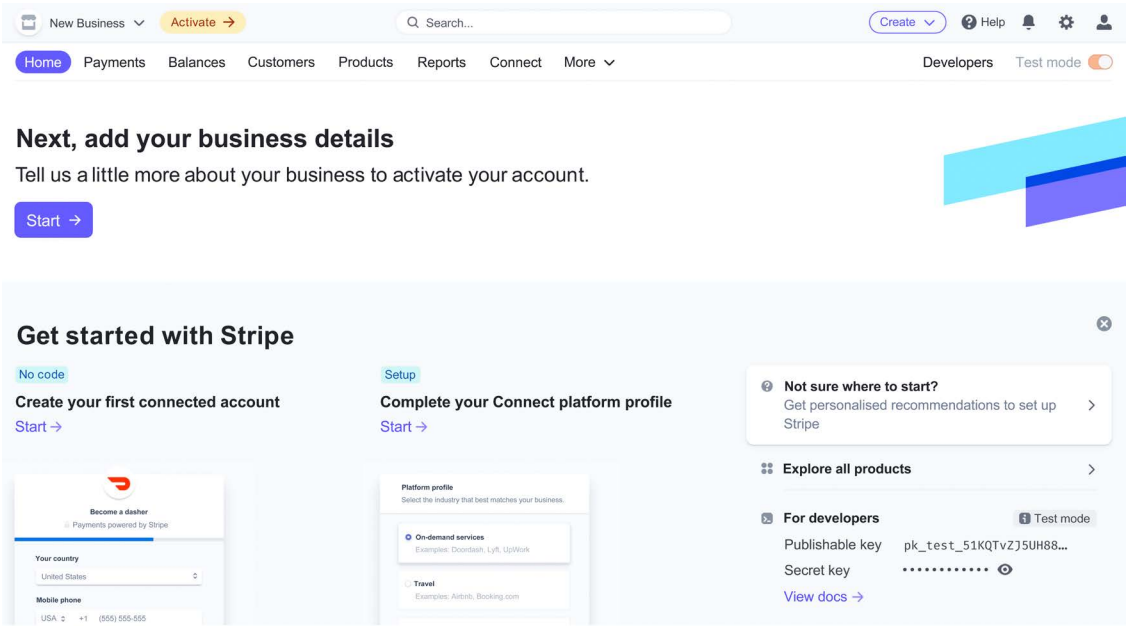


Figure 9.3: The Stripe dashboard after verifying the email address

In the top right of the screen, you can see that **Test mode** is activated. Stripe provides you with a test environment and a production environment. If you own a business or are a freelancer, you can add your business details to activate the account and get access to process real payments. However, this is not necessary to implement and test payments through Stripe, as we will be working on the test environment.

You need to add an account name to process payments. Open `https://dashboard.stripe.com/settings/account` in your browser. You will see the following screen:

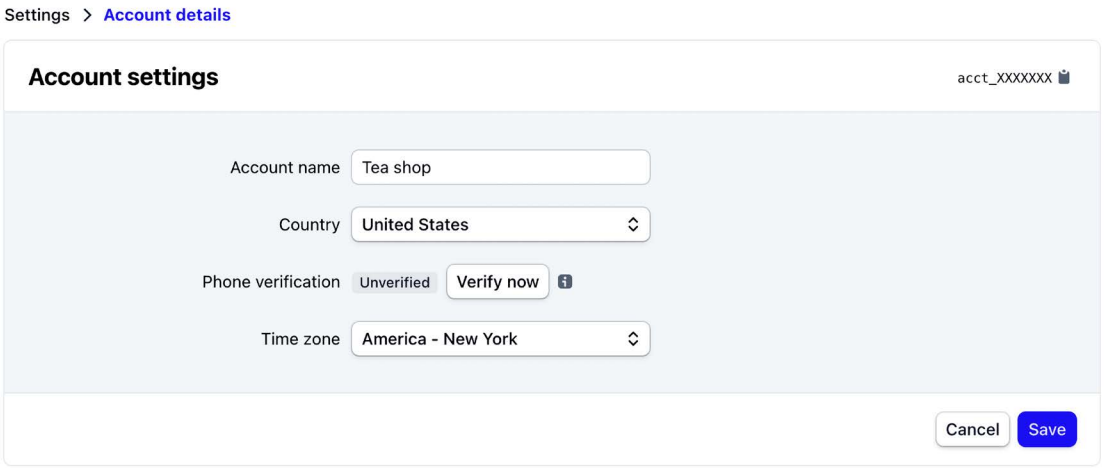


Figure 9.4: The Stripe account settings

Under **Account name**, enter the name of your choice and then click on **Save**. Go back to the Stripe dashboard. You will see your account name displayed in the header:

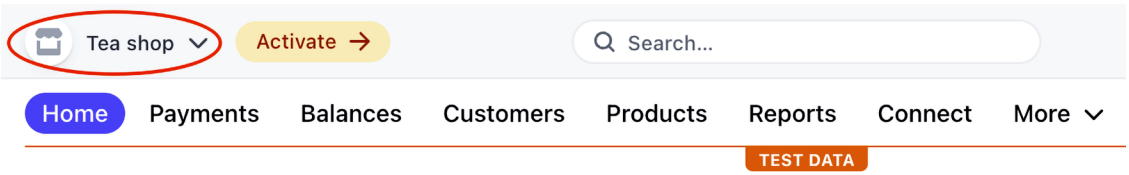


Figure 9.5: The Stripe dashboard header including the account name

We will continue by installing the Stripe Python SDK and adding Stripe to our Django project.

Installing the Stripe Python library

Stripe provides a Python library that simplifies dealing with its API. We are going to integrate the payment gateway into the project using the `stripe` library.

You can find the source code for the Stripe Python library at `https://github.com/stripe/stripe-python`.

Install the `stripe` library from the shell using the following command:

```
pip install stripe==4.0.2
```

Adding Stripe to your project

Open `https://dashboard.stripe.com/test/apikeys` in your browser. You can also access this page from the Stripe dashboard by clicking on **Developers** and then clicking on **API keys**. You will see the following screen:

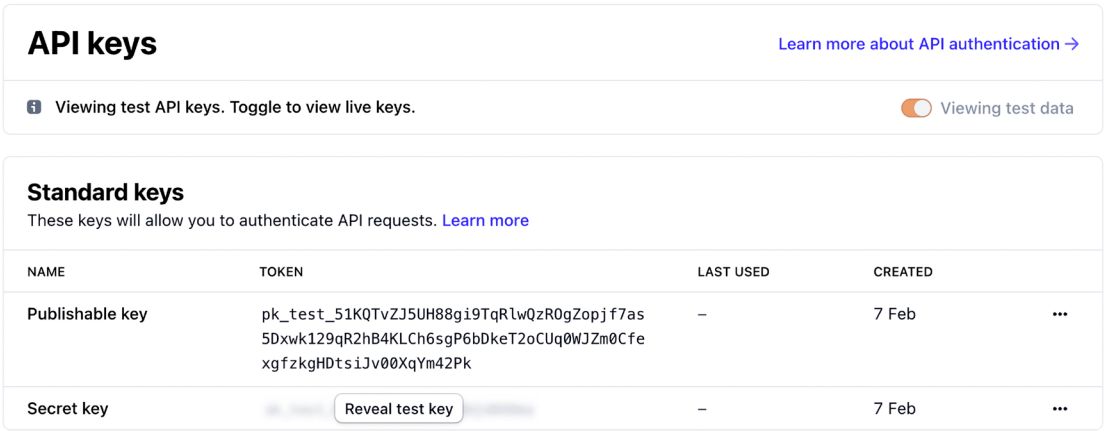


Figure 9.6: The Stripe test API keys screen

Stripe provides a key pair for two different environments, test and production. There is a **Publishable key** and a **Secret key** for each environment. Test mode publishable keys have the prefix `pk_test_` and live mode publishable keys have the prefix `pk_live_`. Test mode secret keys have the prefix `sk_test_` and live mode secret keys have the prefix `sk_live_`.

You will need this information to authenticate requests to the Stripe API. You should always keep your private key secret and store it securely. The publishable key can be used in client-side code such as JavaScript scripts. You can read more about Stripe API keys at `https://stripe.com/docs/keys`.

Add the following settings to the `settings.py` file of your project:

```
# Stripe settings
STRIPE_PUBLISHABLE_KEY = '' # Publishable key
STRIPE_SECRET_KEY = ''      # Secret key
STRIPE_API_VERSION = '2022-08-01'
```

Replace the `STRIPE_PUBLISHABLE_KEY` and `STRIPE_SECRET_KEY` values with the test **Publishable key** and the **Secret key** provided by Stripe. You will use Stripe API version `2022-08-01`. You can see the release notes for this API version at `https://stripe.com/docs/upgrades#2022-08-01`.

You are using the test environment keys for the project. Once you go live and validate your Stripe account, you will obtain the production environment keys. In *Chapter 17*, *Going Live*, you will learn how to configure settings for multiple environments.

Let's integrate the payment gateway into the checkout process. You can find the Python documentation for Stripe at `https://stripe.com/docs/api?lang=python`.

Building the payment process

The checkout process will work as follows:

1. Add items to the shopping cart
2. Check out the shopping cart
3. Enter credit card details and pay

We are going to create a new application to manage payments. Create a new application in your project using the following command:

```
python manage.py startapp payment
```

Edit the `settings.py` file of the project and add the new application to the `INSTALLED_APPS` setting, as follows. The new line is highlighted in bold:

```
INSTALLED_APPS = [
    # ...
    'shop.apps.ShopConfig',
    'cart.apps.CartConfig',
    'orders.apps.OrdersConfig',
    'payment.apps.PaymentConfig',
]
```

The `payment` application is now active in the project.

Currently, users are able to place orders but they cannot pay for them. After clients place an order, we need to redirect them to the payment process.

Edit the `views.py` file of the `orders` application and include the following imports:

```
from django.urls import reverse
from django.shortcuts import render, redirect
```

In the same file, find the following lines of the `order_create` view:

```
# launch asynchronous task
order_created.delay(order.id)
return render(request,
              'orders/order/created.html',
              locals())
```

Replace them with the following code:

```
# launch asynchronous task
order_created.delay(order.id)
# set the order in the session
request.session['order_id'] = order.id
# redirect for payment
return redirect(reverse('payment:process'))
```

The edited view should look as follows:

```
from django.urls import reverse
from django.shortcuts import render, redirect
# ...

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()
            # launch asynchronous task
            order_created.delay(order.id)
            # set the order in the session
            request.session['order_id'] = order.id
            # redirect for payment
            return redirect(reverse('payment:process'))
    else:
        form = OrderCreateForm()
    return render(request,
                  'orders/order/create.html',
                  {'cart': cart, 'form': form})
```

Instead of rendering the template `orders/order/created.html` when placing a new order, the order ID is stored in the user session and the user is redirected to the `payment:process` URL. We are going to implement this URL later. Remember that Celery has to be running for the `order_created` task to be queued and executed.

Let's integrate the payment gateway.

Integrating Stripe Checkout

The Stripe Checkout integration consists of a checkout page hosted by Stripe that allows the user to enter the payment details, usually a credit card, and collects the payment. If the payment is successful, Stripe redirects the client to a success page. If the payment is canceled by the client, it redirects the client to a cancel page.

We will implement three views:

- `payment_process`: Creates a Stripe **Checkout Session** and redirects the client to the Stripe-hosted payment form. A checkout session is a programmatic representation of what the client sees when they are redirected to the payment form, including the products, quantities, currency, and amount to charge
- `payment_completed`: Displays a message for successful payments. The user is redirected to this view if the payment is successful
- `payment_canceled`: Displays a message for canceled payments. The user is redirected to this view if the payment is canceled

Figure 9.7 shows the checkout payment flow:

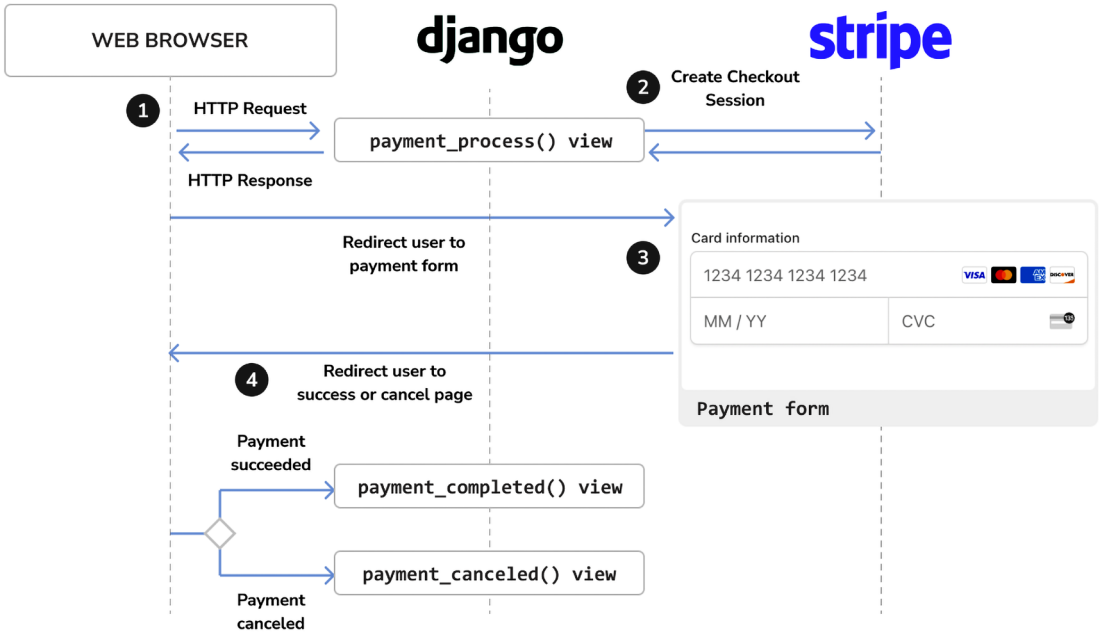


Figure 9.7: The checkout payment flow

The complete checkout process will work as follows:

1. After an order is created, the user is redirected to the `payment_process` view. The user is presented with an order summary and a button to proceed with the payment.
2. When the user proceeds to pay, a Stripe checkout session is created. The checkout session includes the list of items that the user will purchase, a URL to redirect the user to after a successful payment, and a URL to redirect the user to if the payment is canceled.
3. The view redirects the user to the Stripe-hosted checkout page. This page includes the payment form. The client enters their credit card details and submits the form.
4. Stripe processes the payment and redirects the client to the `payment_completed` view. If the client doesn't complete the payment, Stripe redirects the client to the `payment_canceled` view instead.

Let's start building the payment views. Edit the `views.py` file of the `payment` application and add the following code to it:

```
from decimal import Decimal
import stripe
from django.conf import settings
from django.shortcuts import render, redirect, reverse,\
                             get_object_or_404
from orders.models import Order

# create the Stripe instance
stripe.api_key = settings.STRIPE_SECRET_KEY
stripe.api_version = settings.STRIPE_API_VERSION

def payment_process(request):
    order_id = request.session.get('order_id', None)
    order = get_object_or_404(Order, id=order_id)

    if request.method == 'POST':
        success_url = request.build_absolute_uri(
                        reverse('payment:completed'))
        cancel_url = request.build_absolute_uri(
                        reverse('payment:canceled'))
        # Stripe checkout session data
        session_data = {
            'mode': 'payment',
            'client_reference_id': order.id,
            'success_url': success_url,
            'cancel_url': cancel_url,
            'line_items': []
        }
```

```
        # create Stripe checkout session
        session = stripe.checkout.Session.create(**session_data)
        # redirect to Stripe payment form
        return redirect(session.url, code=303)

    else:
        return render(request, 'payment/process.html', locals())
```

In the previous code, the `stripe` module is imported and the Stripe API key is set using the value of the `STRIPE_SECRET_KEY` setting. The API version to use is also set using the value of the `STRIPE_API_VERSION` setting.

The `payment_process` view performs the following tasks:

1. The current `Order` object is retrieved from the database using the `order_id` session key, which was stored previously in the session by the `order_create` view.
2. The `Order` object for the given ID is retrieved. By using the shortcut function `get_object_or_404()`, an `Http404` (page not found) exception is raised if no order is found with the given ID.
3. If the view is loaded with a `GET` request, the template `payment/process.html` is rendered and returned. This template will include the order summary and a button to proceed with the payment, which will generate a `POST` request to the view.
4. If the view is loaded with a `POST` request, a Stripe checkout session is created with `stripe.checkout.Session.create()` using the following parameters:
    - `mode`: The mode of the checkout session. We use `payment` for a one-time payment. You can see the different values accepted for this parameter at `https://stripe.com/docs/api/checkout/sessions/object#checkout_session_object-mode`.
    - `client_reference_id`: The unique reference for this payment. We will use this to reconcile the Stripe checkout session with our order. By passing the order ID, we link Stripe payments to orders in our system, and we will be able to receive payment notifications from Stripe to mark the orders as paid.
    - `success_url`: The URL for Stripe to redirect the user to if the payment is successful. We use `request.build_absolute_uri()` to generate an absolute URI from the URL path. You can see the documentation for this method at `https://docs.djangoproject.com/en/4.1/ref/request-response/#django.http.HttpRequest.build_absolute_uri`.
    - `cancel_url`: The URL for Stripe to redirect the user to if the payment is canceled.
    - `line_items`: This is an empty list. We will next populate it with the order items to be purchased.
5. After creating the checkout session, an HTTP redirect with status code `303` is returned to redirect the user to Stripe. The status code `303` is recommended to redirect web applications to a new URI after an HTTP `POST` has been performed.

You can see all the parameters to create a Stripe session object at `https://stripe.com/docs/api/checkout/sessions/create`.

Let's populate the `line_items` list with the order items to create the checkout session. Each item will contain the name of the item, the amount to charge, the currency to use, and the quantity purchased.

Add the following code highlighted in bold to the `payment_process` view:

```
def payment_process(request):
    order_id = request.session.get('order_id', None)
    order = get_object_or_404(Order, id=order_id)

    if request.method == 'POST':
        success_url = request.build_absolute_uri(
                        reverse('payment:completed'))
        cancel_url = request.build_absolute_uri(
                        reverse('payment:canceled'))
        # Stripe checkout session data
        session_data = {
            'mode': 'payment',
            'success_url': success_url,
            'cancel_url': cancel_url,
            'line_items': []
        }
        # add order items to the Stripe checkout session
        for item in order.items.all():
            session_data['line_items'].append({
                'price_data': {
                    'unit_amount': int(item.price * Decimal('100')),
                    'currency': 'usd',
                    'product_data': {
                        'name': item.product.name,
                    },
                },
                'quantity': item.quantity,
            })
        # create Stripe checkout session
        session = stripe.checkout.Session.create(**session_data)
        # redirect to Stripe payment form
        return redirect(session.url, code=303)

    else:
        return render(request, 'payment/process.html', locals())
```

We use the following information for each item:

- `price_data`: Price-related information.
  - `unit_amount`: The amount in cents to be collected by the payment. This is a positive integer representing how much to charge in the smallest currency unit with no decimal places. For example, to charge $10.00, this would be 1000 (that is, 1,000 cents). The item price, item.price, is multiplied by Decimal('100') to obtain the value in cents and then it is converted into an integer.
  - `currency`: The currency to use in three-letter ISO format. We use usd for US dollars. You can see a list of supported currencies at `https://stripe.com/docs/currencies`.
  - `product_data`: Product-related information.
    - `name`: The name of the product.
- `quantity`: The number of units to purchase.

The `payment_process` view is now ready. Let's create simple views for the payment success and cancel pages.

Add the following code to the `views.py` file of the `payment` application:

```
def payment_completed(request):
    return render(request, 'payment/completed.html')

def payment_canceled(request):
    return render(request, 'payment/canceled.html')
```

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

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

app_name = 'payment'

urlpatterns = [
    path('process/', views.payment_process, name='process'),
    path('completed/', views.payment_completed, name='completed'),
    path('canceled/', views.payment_canceled, name='canceled'),
]
```

These are the URLs for the payment workflow. We have included the following URL patterns:

- `process`: The view that displays the order summary to the user, creates the Stripe checkout session, and redirects the user to the Stripe-hosted payment form

- `completed`: The view for Stripe to redirect the user to if the payment is successful
- `canceled`: The view for Stripe to redirect the user to if the payment is canceled

Edit the main `urls.py` file of the `myshop` project and include the URL patterns for the `payment` application, as follows:

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

We have placed the new path before the `shop.urls` pattern to avoid an unintended pattern match with a pattern defined in `shop.urls`. Remember that Django runs through each URL pattern in order and stops at the first one that matches the requested URL.

Let's build a template for each view. Create the following file structure inside the `payment` application directory:

```
templates/
    payment/
        process.html
        completed.html
        canceled.html
```

Edit the `payment/process.html` template and add the following code to it:

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

{% block title %}Pay your order{% endblock %}

{% block content %}
  <h1>Order summary</h1>
  <table class="cart">
    <thead>
      <tr>
        <th>Image</th>
        <th>Product</th>
        <th>Price</th>
        <th>Quantity</th>
        <th>Total</th>
      </tr>
    </thead>
```

```
    <tbody>
      {% for item in order.items.all %}
        <tr class="row{% cycle "1" "2" %}">
          <td>
            <img src="{% if item.product.image %}{{ item.product.image.url }}
            {% else %}{% static "img/no_image.png" %}{% endif %}">
          </td>
          <td>{{ item.product.name }}</td>
          <td class="num">${{ item.price }}</td>
          <td class="num">{{ item.quantity }}</td>
          <td class="num">${{ item.get_cost }}</td>
        </tr>
      {% endfor %}
      <tr class="total">
        <td colspan="4">Total</td>
        <td class="num">${{ order.get_total_cost }}</td>
      </tr>
    </tbody>
  </table>
  <form action="{% url "payment:process" %}" method="post">
    <input type="submit" value="Pay now">
    {% csrf_token %}
  </form>
{% endblock %}
```

This is the template to display the order summary to the user and allow the client to proceed with the payment. It includes a form and a **Pay now** button to submit it via `POST`. When the form is submitted, the `payment_process` view creates the Stripe checkout session and redirects the user to the Stripe-hosted payment form.

Edit the `payment/completed.html` template and add the following code to it:

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

{% block title %}Payment successful{% endblock %}

{% block content %}
  <h1>Your payment was successful</h1>
  <p>Your payment has been processed successfully.</p>
{% endblock %}
```

This is the template for the page that the user is redirected to after a successful payment.

Edit the `payment/canceled.html` template and add the following code to it:

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

{% block title %}Payment canceled{% endblock %}

{% block content %}
  <h1>Your payment has not been processed</h1>
  <p>There was a problem processing your payment.</p>
{% endblock %}
```

This is the template for the page that the user is redirected to when the payment is canceled.

We have implemented the necessary views to process payments, including their URL patterns and templates. It's time to try out the checkout process.

Testing the checkout process

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
rabbitmq:management
```

This will run RabbitMQ on port `5672` and the web-based management interface on port `15672`.

Open another shell and start the Celery worker from your project directory with the following command:

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

Open one more shell and start the development server from your project directory with this command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/` in your browser, add some products to the shopping cart, and fill in the checkout form. Click the **Place order** button. The order will be persisted to the database, the order ID will be saved in the current session, and you will be redirected to the payment process page.

The payment process page will look as follows:

My shop

Order summary

Image	Product	Price	Quantity	Total
	Green tea	$30.00	1	$30.00
	Red tea	$45.50	2	$91.00
Total				$121.00

Pay now

Figure 9.8: The payment process page including an order summary

Images in this chapter:

- *Green tea*: Photo by Jia Ye on Unsplash
- *Red tea*: Photo by Manki Kim on Unsplash

On this page, you can see an order summary and a **Pay now** button. Click on **Pay now**. The `payment_process` view will create a Stripe checkout session and you will be redirected to the Stripe-hosted payment form. You will see the following page:

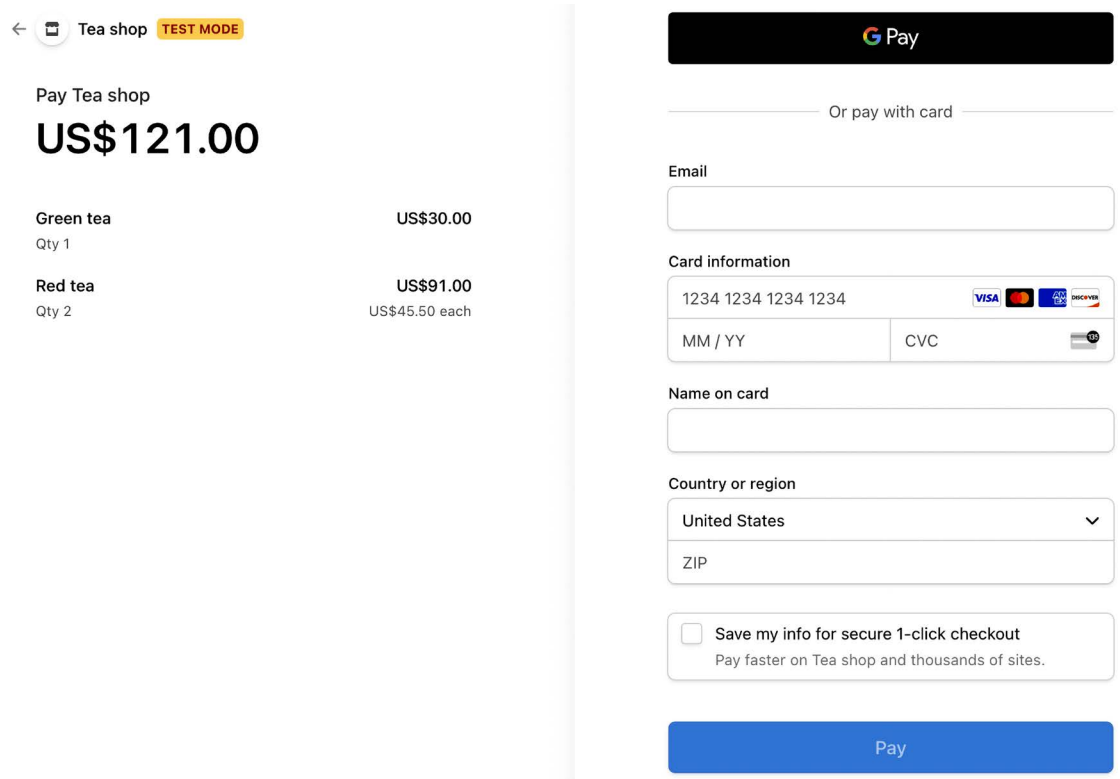


Figure 9.9: The Stripe checkout payment from

Using test credit cards

Stripe provides different test credit cards from different card issuers and countries, which allows you to simulate payments to test all possible scenarios (successful payment, declined payment, etc.). The following table shows some of the cards you can test for different scenarios:

Result	Test Credit Card	CVC	Expiry date
Successful payment	`4242 4242 4242 4242`	Any 3 digits	Any future date
Failed payment	`4000 0000 0000 0002`	Any 3 digits	Any future date
Requires 3D secure authentication	`4000 0025 0000 3155`	Any 3 digits	Any future date

You can find the complete list of credit cards for testing at `https://stripe.com/docs/testing`.

We are going to use the test card `4242 4242 4242 4242`, which is a Visa card that returns a successful purchase. We will use the CVC `123` and any future expiration date, such as `12/29`. Enter the credit card details in the payment form as follows:

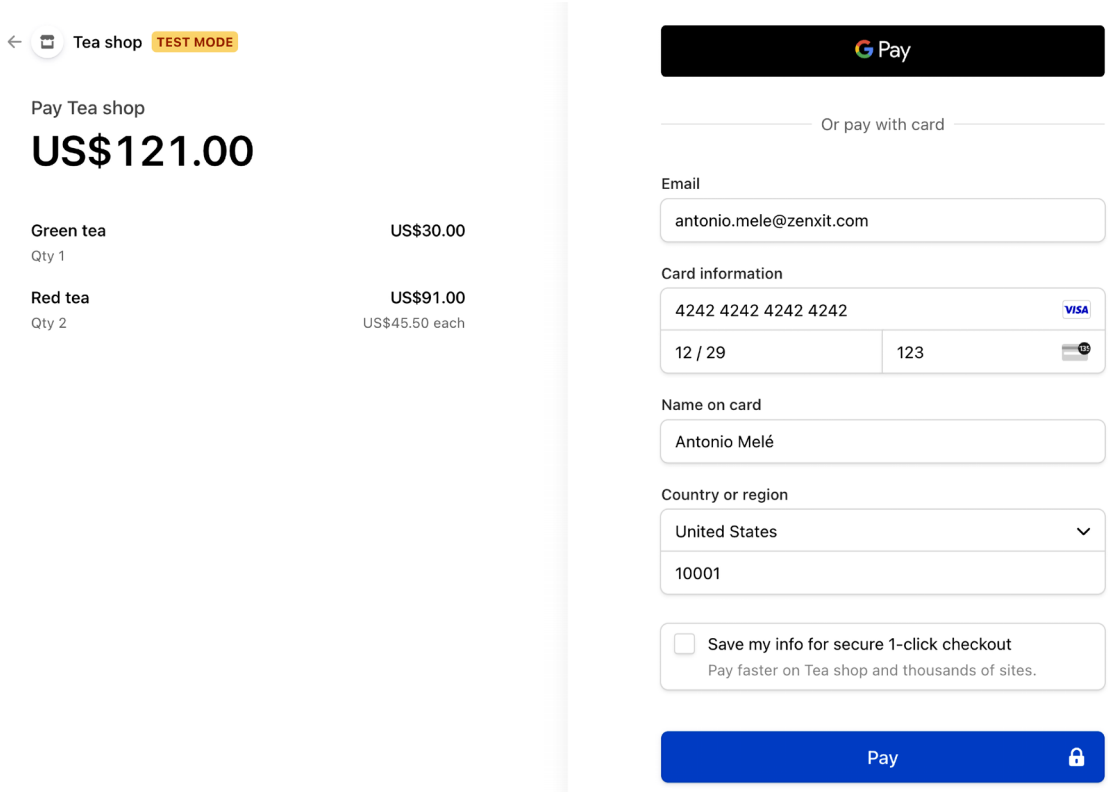


Figure 9.10: The payment form with the valid test credit card details

Click the **Pay** button. The button text will change to **Processing...**, as in *Figure 9.11*:

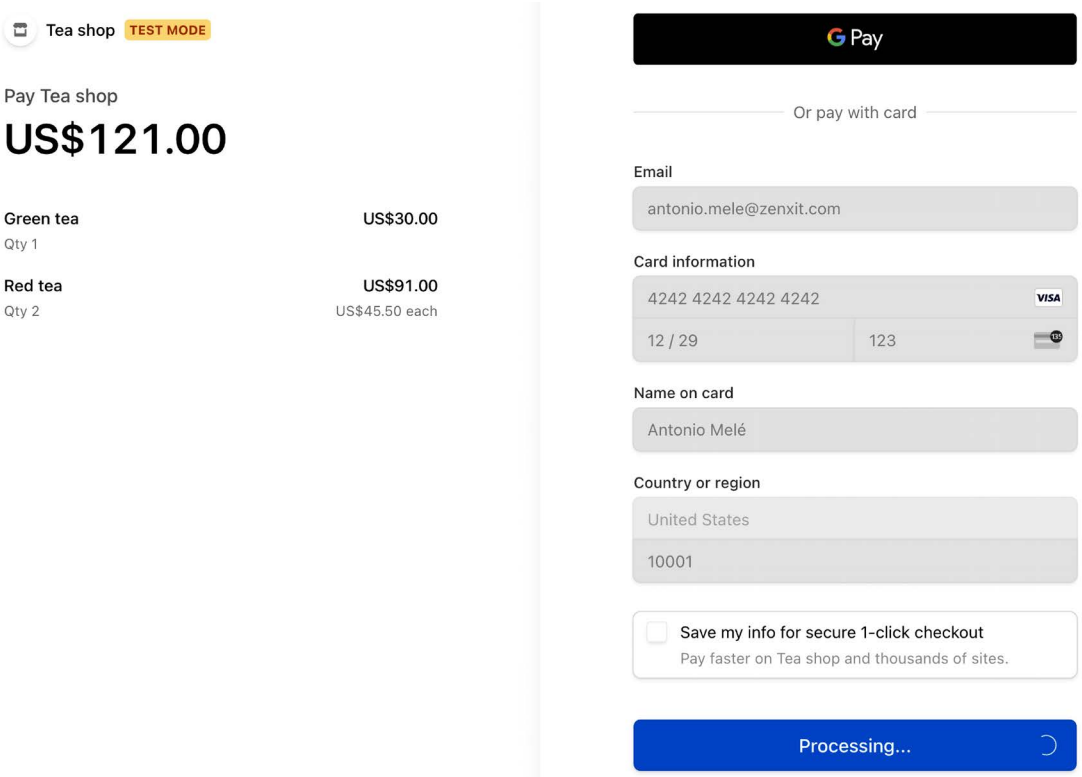


Figure 9.11: The payment form being processed

After a couple of seconds, you will see the button turns green like in *Figure 9.12*:

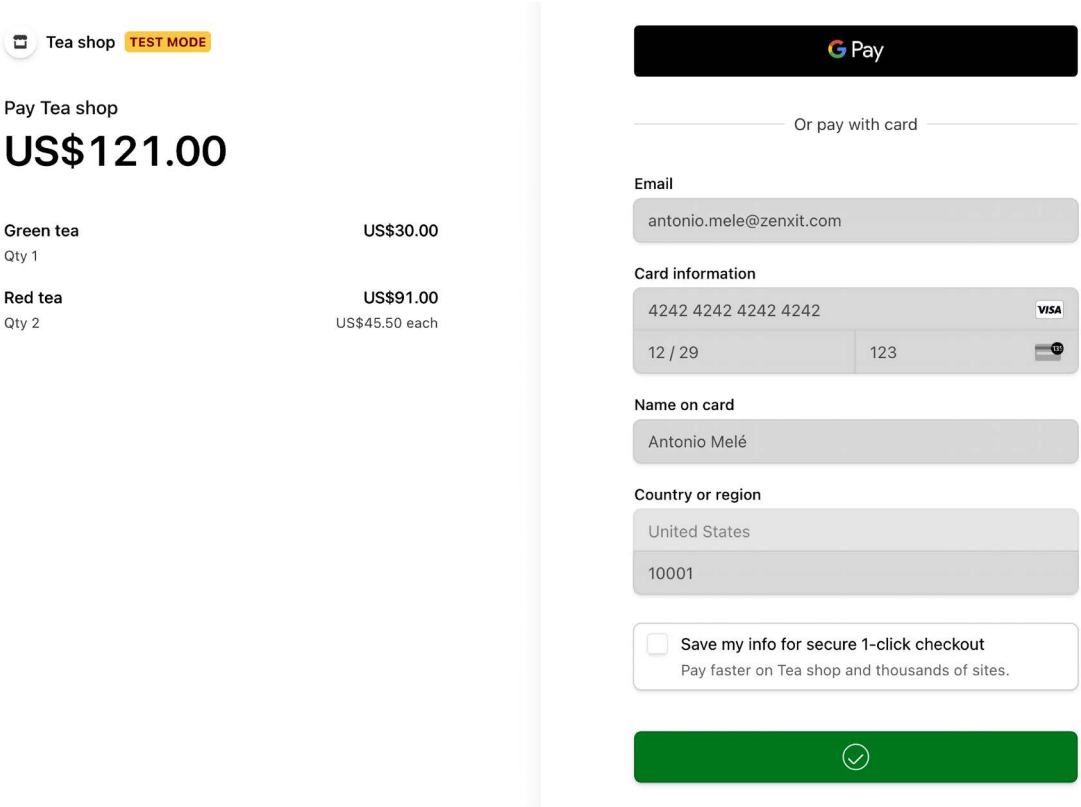


Figure 9.12: The payment form after the payment is successful

Then Stripe redirects your browser to the payment completed URL you provided when creating the checkout session. You will see the following page:

Figure 9.13: The successful payment page

Checking the payment information in the Stripe dashboard

Access the Stripe dashboard at `https://dashboard.stripe.com/test/payments`. Under **Payments**, you will be able to see the payment like in *Figure 9.14*:

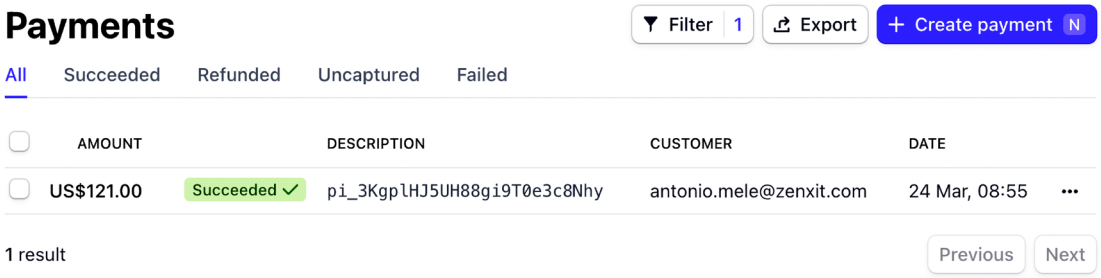


Figure 9.14: The payment object with status Succeeded in the Stripe dashboard

The payment status is **Succeeded**. The payment description includes the **payment intent** ID that starts with `pi_`. When a checkout session is confirmed, Stripe creates a payment intent associated with the session. A payment intent is used to collect a payment from the user. Stripe records all attempted payments as payment intents. Each payment intent has a unique ID, and it encapsulates the details of the transaction, such as the supported payment methods, the amount to collect, and the desired currency. Click on the transaction to access the payment details.

You will see the following screen:

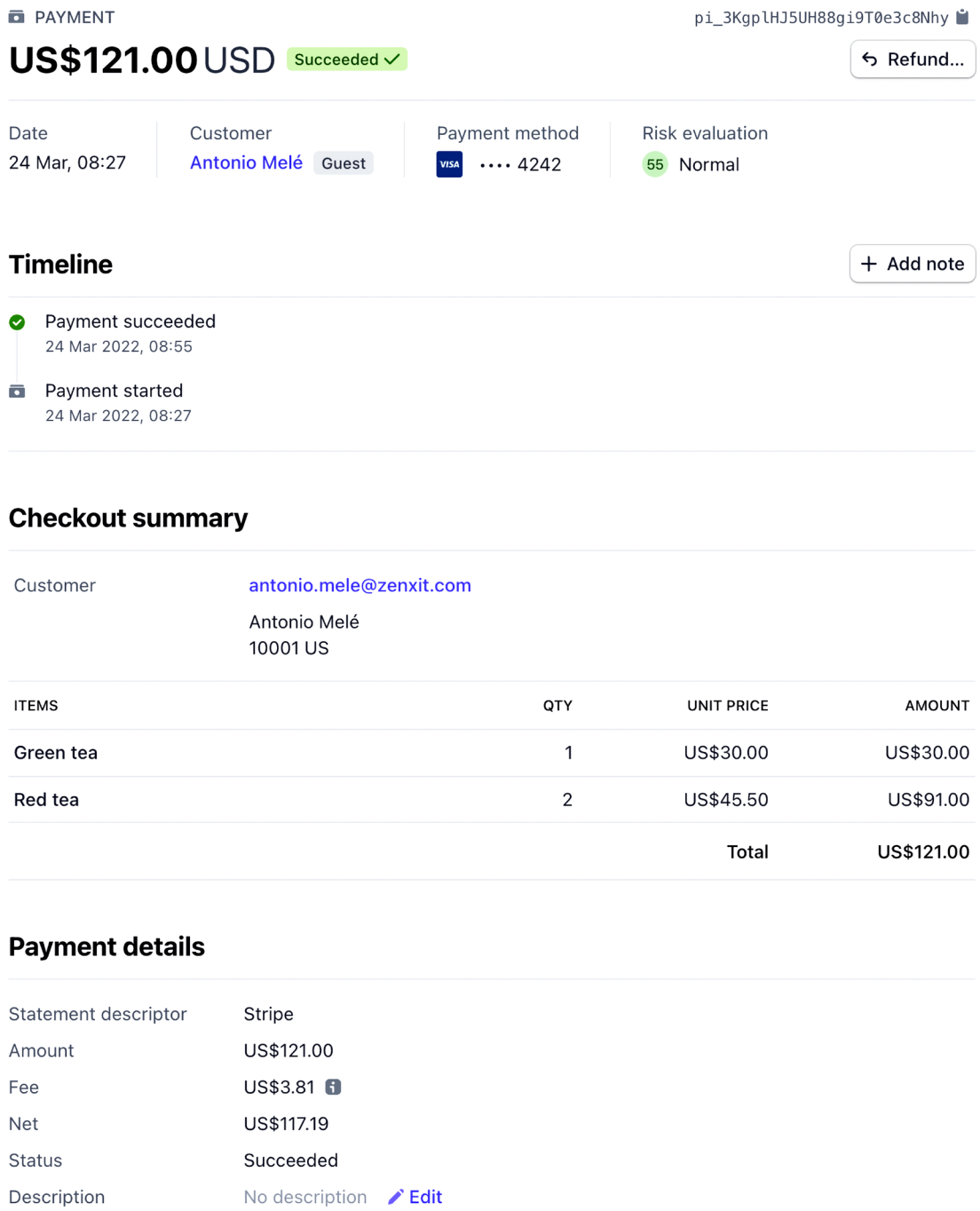


Figure 9.15: Payment details for a Stripe transaction

Here you can see the payment information and the payment timeline, including payment changes. Under **Checkout summary**, you can find the line items purchased, including name, quantity, unit price, and amount. Under **Payment details**, you can see a breakdown of the amount paid and the Stripe fee for processing the payment.

Under this section, you will find a **Payment method** section including details about the payment method and the credit card checks performed by Stripe, like in *Figure 9.16*:

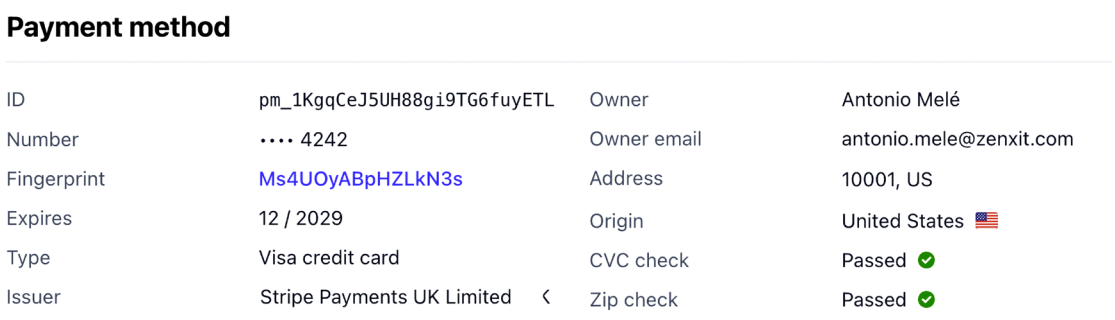


Figure 9.16: Payment method used in the Stripe transaction

Under this section, you will find another section named **Events and logs**, like in *Figure 9.17*:

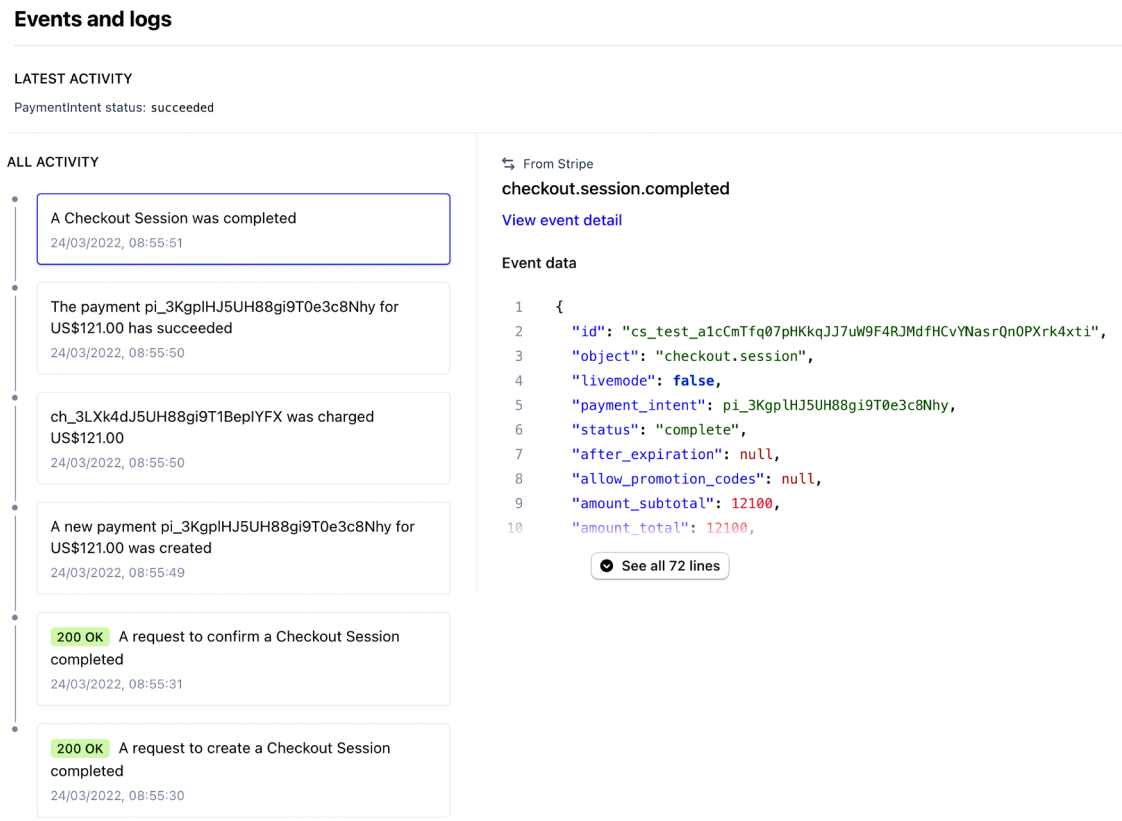


Figure 9.17: Events and logs for a Stripe transaction

This section contains all the activity related to the transaction, including requests to the Stripe API. You can click on any request to see the HTTP request to the Stripe API and the response in JSON format.

Let's review the activity events in chronological order, from bottom to top:

1. First, a new checkout session is created by sending a `POST` request to the Stripe API endpoint `/v1/checkout/sessions`. The Stripe SDK method `stripe.checkout.Session.create()` that is used in the `payment_process` view builds and sends the request to the Stripe API and handles the response to return a session object.
2. The user is redirected to the checkout page where they submit the payment form. A request to confirm the checkout session is sent by the Stripe checkout page.
3. A new payment intent is created.
4. A charge related to the payment intent is created.
5. The payment intent is now completed with a successful payment.
6. The checkout session is completed.

Congratulations! You have successfully integrated Stripe Checkout into your project. Next, you will learn how to receive payment notifications from Stripe and how to reference Stripe payments in your shop orders.

Using webhooks to receive payment notifications

Stripe can push real-time events to our application by using webhooks. A **webhook**, also called a callback, can be thought of as an event-driven API instead of a request-driven API. Instead of polling the Stripe API frequently to know when a new payment is completed, Stripe can send an HTTP request to a URL of our application to notify of successful payments in real time. These notification of these events will be asynchronous, when the event occurs, regardless of our synchronous calls to the Stripe API.

We will build a webhook endpoint to receive Stripe events. The webhook will consist of a view that will receive a JSON payload with the event information to process it. We will use the event information to mark orders as paid when the checkout session is successfully completed.

Creating a webhook endpoint

You can add webhook endpoint URLs to your Stripe account to receive events. Since we are using webhooks and we don't have a hosted website accessible through a public URL, we will use the Stripe **Command-Line Interface** (**CLI**) to listen to events and forward them to our local environment.

Open `https://dashboard.stripe.com/test/webhooks` in your browser. You will see the following screen:

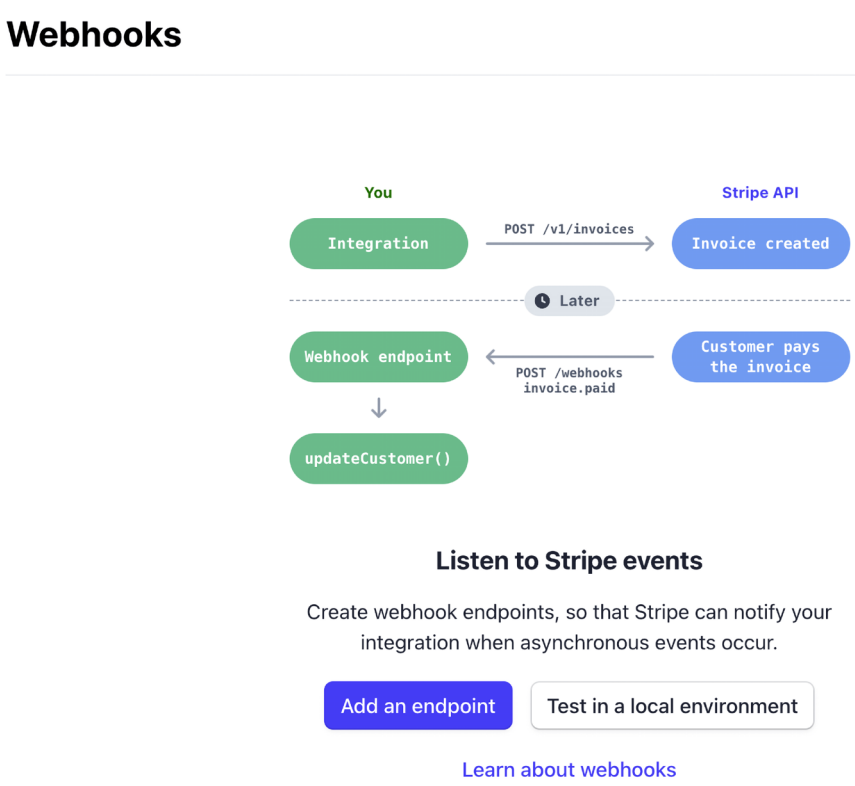


Figure 9.18: The Stripe webhooks default screen

Here you can see a schema of how Stripe notifies your integration asynchronously. You will get Stripe notifications in real time whenever an event happens. Stripe sends different types of events like checkout session created, payment intent created, payment intent updated, or checkout session completed. You can find a list of all the types of events that Stripe sends at `https://stripe.com/docs/api/events/types`.

Click on **Test in a local environment**. You will see the following screen:

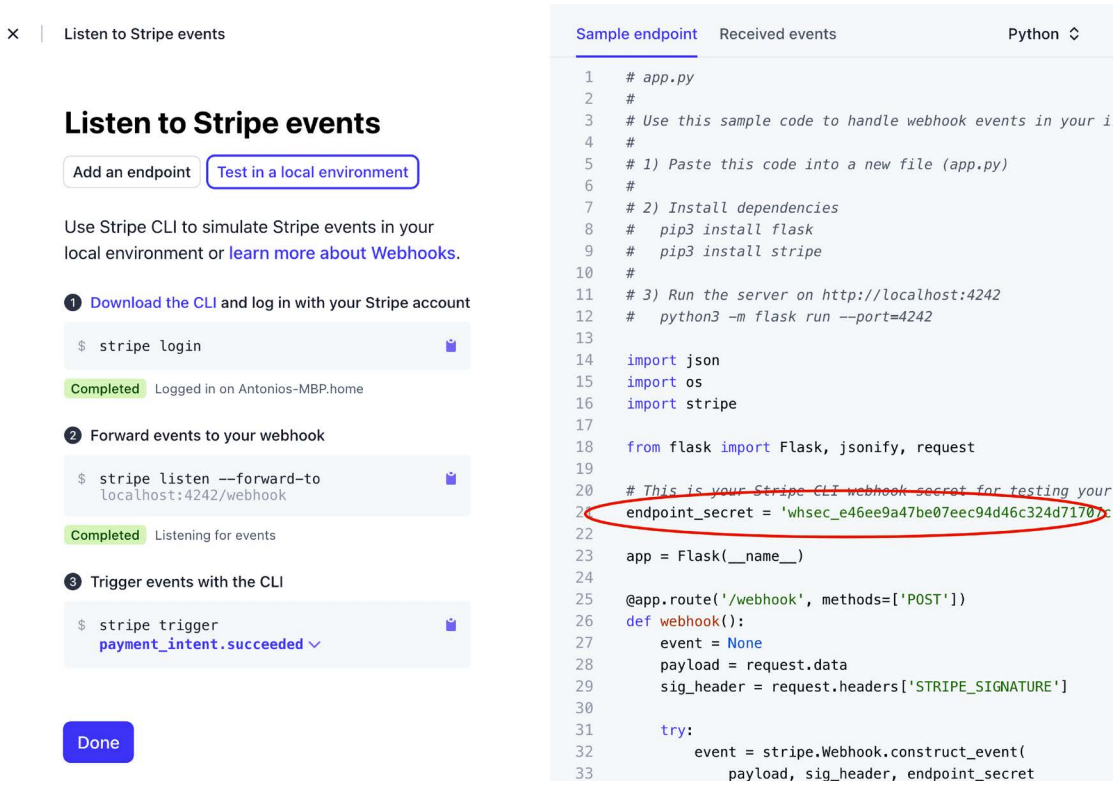


Figure 9.19: The Stripe webhook setup screen

This screen shows the steps to listen to Stripe events from your local environment. It also includes a sample Python webhook endpoint. Copy just the `endpoint_secret` value.

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

```
STRIPE_WEBHOOK_SECRET = ''
```

Replace the `STRIPE_WEBHOOK_SECRET` value with the `endpoint_secret` value provided by Stripe.

To build a webhook endpoint, we will create a view that receives a JSON payload with the event details. We will check the event details to identify when a checkout session is completed and mark the related order as paid.

Stripe signs the webhook events it sends to your endpoints by including a `Stripe-Signature` header with a signature in each event. By checking the Stripe signature, you can verify that events were sent by Stripe and not by a third party. If you don't check the signature, an attacker could send fake events to your webhooks intentionally. The Stripe SDK provides a method to verify signatures. We will use it to create a webhook that verifies the signature.

Add a new file to the `payment/` application directory and name it `webhooks.py`. Add the following code to the new `webhooks.py` file:

```
import stripe
from django.conf import settings
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from orders.models import Order


@csrf_exempt
def stripe_webhook(request):
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
    event = None

    try:
        event = stripe.Webhook.construct_event(
                    payload,
                    sig_header,
                    settings.STRIPE_WEBHOOK_SECRET)
    except ValueError as e:
        # Invalid payload
        return HttpResponse(status=400)
    except stripe.error.SignatureVerificationError as e:
        # Invalid signature
        return HttpResponse(status=400)

    return HttpResponse(status=200)
```

The `@csrf_exempt` decorator is used to prevent Django from performing the CSRF validation that is done by default for all `POST` requests. We use the method `stripe.Webhook.construct_event()` of the `stripe` library to verify the event's signature header. If the event's payload or the signature is invalid, we return an HTTP `400 Bad Request` response. Otherwise, we return an HTTP `200 OK` response. This is the basic functionality required to verify the signature and construct the event from the JSON payload. Now we can implement the actions of the webhook endpoint.

Add the following code highlighted in bold to the `stripe_webhook` view:

```
@csrf_exempt
def stripe_webhook(request):
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
```

```
    event = None

    try:
        event = stripe.Webhook.construct_event(
                    payload,
                    sig_header,
                    settings.STRIPE_WEBHOOK_SECRET)
    except ValueError as e:
        # Invalid payload
        return HttpResponse(status=400)
    except stripe.error.SignatureVerificationError as e:
        # Invalid signature
        return HttpResponse(status=400)

    if event.type == 'checkout.session.completed':
        session = event.data.object
        if session.mode == 'payment' and session.payment_status == 'paid':
            try:
                order = Order.objects.get(id=session.client_reference_id)
            except Order.DoesNotExist:
                return HttpResponse(status=404)
            # mark order as paid
            order.paid = True
            order.save()

    return HttpResponse(status=200)
```

In the new code, we check if the event received is `checkout.session.completed`. This event indicates that the checkout session has been successfully completed. If we receive this event, we retrieve the session object and check whether the session `mode` is `payment` because this is the expected mode for one-off payments. Then we get the `client_reference_id` attribute that we used when we created the checkout session and use the Django ORM to retrieve the `Order` object with the given `id`. If the order does not exist, we raise an HTTP `404` exception. Otherwise, we mark the order as paid with `order.paid = True` and we save the order to the database.

Edit the `urls.py` file of the `payment` application and add the following code highlighted in bold:

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

app_name = 'payment'

urlpatterns = [
    path('process/', views.payment_process, name='process'),
    path('completed/', views.payment_completed, name='completed'),
    path('canceled/', views.payment_canceled, name='canceled'),
    path('webhook/', webhooks.stripe_webhook, name='stripe-webhook'),
]
```

We have imported the webhooks module and added the URL pattern for the Stripe webhook.

Testing webhook notifications

To test webhooks, you need to install the Stripe CLI. The Stripe CLI is a developer tool that allows you to test and manage your integration with Stripe directly from your shell. You will find installation instructions at `https://stripe.com/docs/stripe-cli#install`.

If you are using macOS or Linux, you can install the Stripe CLI with Homebrew using the following command:

```
brew install stripe/stripe-cli/stripe
```

If you are using Windows, or you are using macOS or Linux without Homebrew, download the latest Stripe CLI release for macOS, Linux, or Windows from `https://github.com/stripe/stripe-cli/releases/latest` and unzip the file. If you are using Windows, run the unzipped `.exe` file.

After installing the Stripe CLI, run the following command from a shell:

```
stripe login
```

You will see the following output:

```
Your pairing code is: xxxx-yyyy-zzzz-oooo
This pairing code verifies your authentication with Stripe.
Press Enter to open the browser or visit https://dashboard.stripe.com/
stripecli/confirm_auth?t=....
```

Press *Enter* or open the URL in your browser. You will see the following screen:

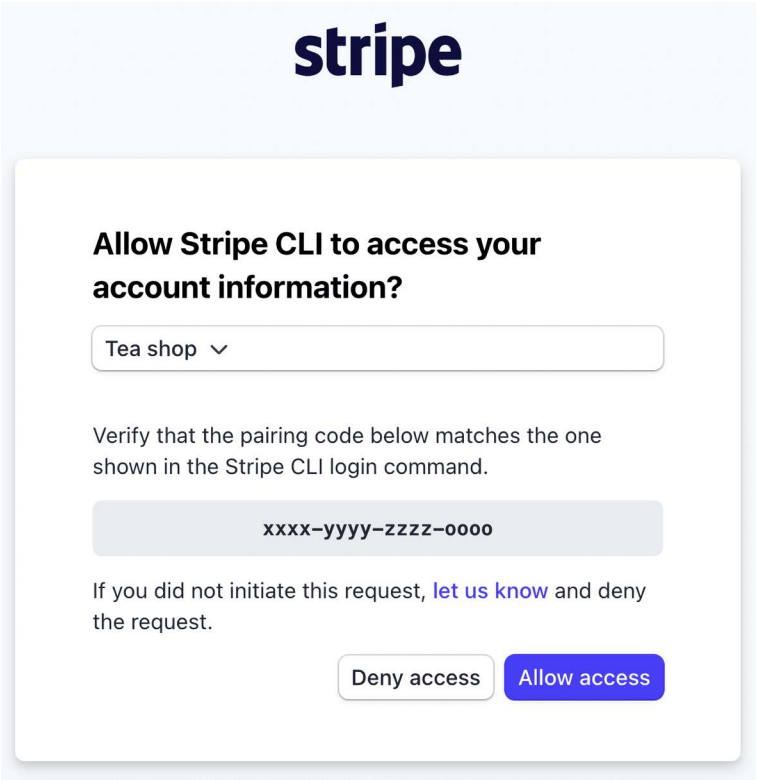


Figure 9.20: The Stripe CLI pairing screen

Verify that the pairing code in the Stripe CLI matches the one shown on the website and click on **Allow access**. You will see the following message:

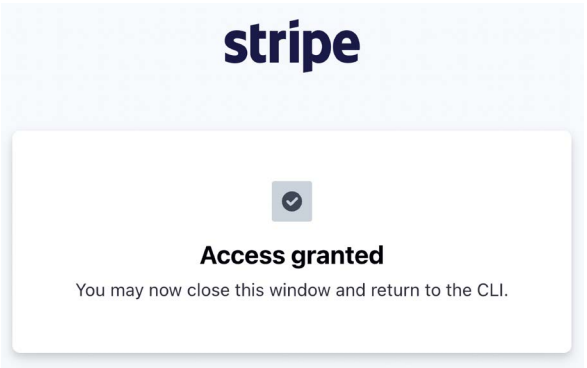


Figure 9.21: The Stripe CLI pairing confirmation

Now run the following command from your shell:

```
stripe listen --forward-to localhost:8000/payment/webhook/
```

We use this command to tell Stripe to listen to events and forward them to our local host. We use port `8000`, where the Django development server is running, and the path `/payment/webhook/`, which matches the URL pattern of our webhook.

You will see the following output:

```
Getting ready... > Ready! You are using Stripe API Version [2022-08-01]. Your
webhook signing secret is xxxxxxxxxxxxxxxxxxx (^C to quit)
```

Here, you can see the webhook secret. Check that the webhook signing secret matches the `STRIPE_WEBHOOK_SECRET` setting in the `settings.py` file of your project.

Open `https://dashboard.stripe.com/test/webhooks` in your browser. You will see the following screen:

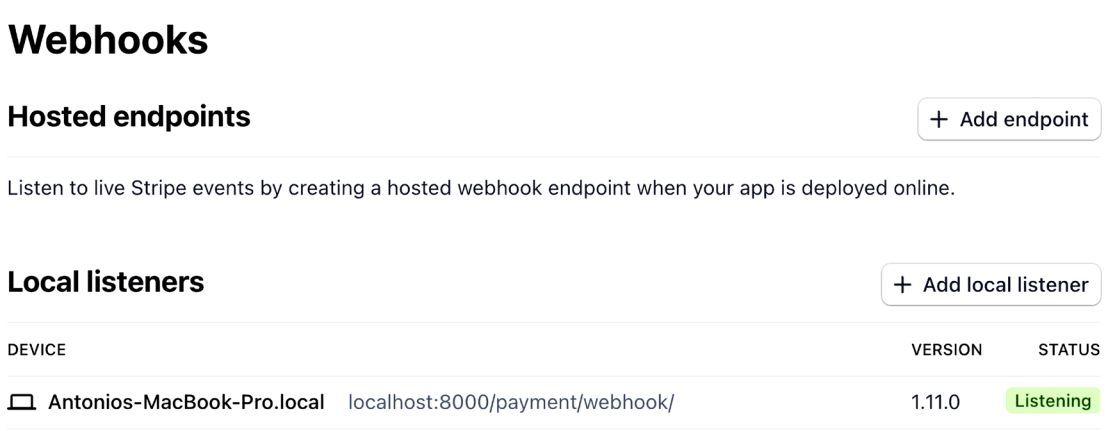


Figure 9.22: The Stripe Webhooks page

Under **Local listeners**, you will see the local listener that we created.

In a production environment, the Stripe CLI is not needed. Instead, you would need to add a hosted webhook endpoint using the URL of your hosted application.

Open `http://127.0.0.1:8000/` in your browser, add some products to the shopping cart, and complete the checkout process.

Check the shell where you are running the Stripe CLI:

```
2022-08-17 13:06:13   --> payment_intent.created [evt_...]
2022-08-17 13:06:13  <--  [200] POST http://localhost:8000/payment/webhook/
[evt_...]

2022-08-17 13:06:13   --> payment_intent.succeeded [evt_...]
2022-08-17 13:06:13  <--  [200] POST http://localhost:8000/payment/webhook/
[evt_...]

2022-08-17 13:06:13   --> charge.succeeded [evt_...]
2022-08-17 13:06:13  <--  [200] POST http://localhost:8000/payment/webhook/
[evt_...]

2022-08-17 13:06:14   --> checkout.session.completed [evt_...]
2022-08-17 13:06:14  <--  [200] POST http://localhost:8000/payment/webhook/
[evt_...]
```

You can see the different events that have been sent by Stripe to the local webhook endpoint. These are, in chronological order:

- `payment_intent.created`: The payment intent has been created.
- `payment_intent.succeeded`: The payment intent succeeded.
- `charge.succeeded`: The charge associated with the payment intent succeeded.
- `checkout.session.completed`: The checkout session has been completed. This is the event that we use to mark the order as paid.

The `stripe_webhook` webhook returns an HTTP `200 OK` response to all of the requests sent by Stripe. However, we only process the event `checkout.session.completed` to mark the order related to the payment as paid.

Next, open `http://127.0.0.1:8000/admin/orders/order/` in your browser. The order should now be marked as paid:

Figure 9.23: An order marked as paid in the order list of the administration site

Now orders get automatically marked as paid with Stripe payment notifications. Next, you are going to learn how to reference Stripe payments in your shop orders.

Referencing Stripe payments in orders

Each Stripe payment has a unique identifier. We can use the payment ID to associate each order with its corresponding Stripe payment. We will add a new field to the `Order` model of the `orders` application, so that we can reference the related payment by its ID. This will allow us to link each order with the related Stripe transaction.

Edit the `models.py` file of the `orders` application and add the following field to the `Order` model. The new field is highlighted in bold:

```
class Order(models.Model):
    # ...
    stripe_id = models.CharField(max_length=250, blank=True)
```

Let's sync this field with the database. Use the following command to generate the database migrations for the project:

```
python manage.py makemigrations
```

You will see the following output:

```
Migrations for 'orders':
  orders/migrations/0002_order_stripe_id.py
    - Add field stripe_id to order
```

Apply the migration to the database with the following command:

```
python manage.py migrate
```

You will see output that ends with the following line:

```
Applying orders.0002_order_stripe_id... OK
```

The model changes are now synced with the database. Now you will be able to store the Stripe payment ID for each order.

Edit the `stripe_webhook` function in the `views.py` file of the payment application and add the following lines highlighted in bold:

```
# ...
@csrf_exempt
def stripe_webhook(request):
    # ...

    if event.type == 'checkout.session.completed':
        session = event.data.object
        if session.mode == 'payment' and session.payment_status == 'paid':
            try:
```

```
                order = Order.objects.get(id=session.client_reference_id)
            except Order.DoesNotExist:
                return HttpResponse(status=404)
            # mark order as paid
            order.paid = True
            # store Stripe payment ID
            order.stripe_id = session.payment_intent
            order.save()
            # launch asynchronous task
            payment_completed.delay(order.id)

    return HttpResponse(status=200)
```

With this change, when receiving a webhook notification for a completed checkout session, the payment intent ID is stored in the `stripe_id` field of the `order` object.

Open `http://127.0.0.1:8000/` in your browser, add some products to the shopping cart, and complete the checkout process. Then, access `http://127.0.0.1:8000/admin/orders/order/` in your browser and click on the latest order ID to edit it. The `stripe_id` field should contain the payment intent ID as in *Figure 9.24*:

Figure 9.24: The Stripe ID field with the payment intent ID

Great! We are successfully referencing Stripe payments in orders. Now, we can add Stripe payment IDs to the order list on the administration site. We can also include a link to each payment ID to see the payment details in the Stripe dashboard.

Edit the `models.py` file of the `orders` application and add the following code highlighted in bold:

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

class Order(models.Model):
    # ...

    class Meta:
        # ...

    def __str__(self):
```

```
        return f'Order {self.id}'

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

    def get_stripe_url(self):
        if not self.stripe_id:
            # no payment associated
            return ''
        if '_test_' in settings.STRIPE_SECRET_KEY:
            # Stripe path for test payments
            path = '/test/'
        else:
            # Stripe path for real payments
            path = '/'
        return f'https://dashboard.stripe.com{path}payments/{self.stripe_id}'
```

We have added the new `get_stripe_url()` method to the `Order` model. This method is used to return the Stripe dashboard's URL for the payment associated with the order. If no payment ID is stored in the `stripe_id` field of the `Order` object, an empty string is returned. Otherwise, the URL for the payment in the Stripe dashboard is returned. We check if the string `_test_` is present in the `STRIPE_SECRET_KEY` setting to discriminate the production environment from the test environment. Payments in the production environment follow the pattern `https://dashboard.stripe.com/payments/{id}`, whereas test payments follow the pattern `https://dashboard.stripe.com/payments/test/{id}`.

Let's add a link to each `Order` object on the list display page of the administration site.

Edit the `admin.py` file of the `orders` application and add the following code highlighted in bold:

```
from django.utils.safestring import mark_safe

def order_payment(obj):
    url = obj.get_stripe_url()
    if obj.stripe_id:
        html = f'<a href="{url}" target="_blank">{obj.stripe_id}</a>'
        return mark_safe(html)
    return ''
order_payment.short_description = 'Stripe payment'

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
```

```
                    'address', 'postal_code', 'city', 'paid',
                    order_payment, 'created', 'updated']
    # ...
```

The `order_stripe_payment()` function takes an `Order` object as an argument and returns an HTML link with the payment URL in Stripe. Django escapes HTML output by default. We use the `mark_safe` function to avoid auto-escaping.

Avoid using `mark_safe` on input that has come from the user to avoid **Cross-Site Scripting (XSS)**. XSS enables attackers to inject client-side scripts into web content viewed by other users.

Open `http://127.0.0.1:8000/admin/orders/order/` in your browser. You will see a new column named **STRIPE PAYMENT**. You will see the related Stripe payment ID for the latest order. If you click on the payment ID, you will be taken to the payment URL in Stripe, where you can find the additional payment details.

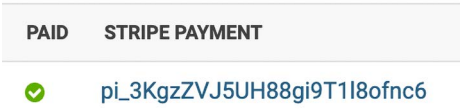


Figure 9.25: The Stripe payment ID for an order object in the administration site

Now you automatically store Stripe payment IDs in orders when receiving payment notifications. You have successfully integrated Stripe into your project.

Going live

Once you have tested your integration, you can apply for a production Stripe account. When you are ready to move into production, remember to replace your test Stripe credentials with the live ones in the `settings.py` file. You will also need to add a webhook endpoint for your hosted website at `https://dashboard.stripe.com/webhooks` instead of using the Stripe CLI. *Chapter 17*, *Going Live*, will teach you how to configure project settings for multiple environments.

Exporting orders to CSV files

Sometimes, you might want to export the information contained in a model to a file so that you can import it into another system. One of the most widely used formats to export/import data is **Comma-Separated Values** (**CSV**). A CSV file is a plain text file consisting of a number of records. There is usually one record per line and some delimiter character, usually a literal comma, separating the record fields. We are going to customize the administration site to be able to export orders to CSV files.

Adding custom actions to the administration site

Django offers a wide range of options to customize the administration site. You are going to modify the object list view to include a custom administration action. You can implement custom administration actions to allow staff users to apply actions to multiple elements at once in the change list view.

An administration action works as follows: a user selects objects from the administration object list page with checkboxes, then they select an action to perform on all of the selected items, and execute the actions. *Figure 9.26* shows where actions are located in the administration site:

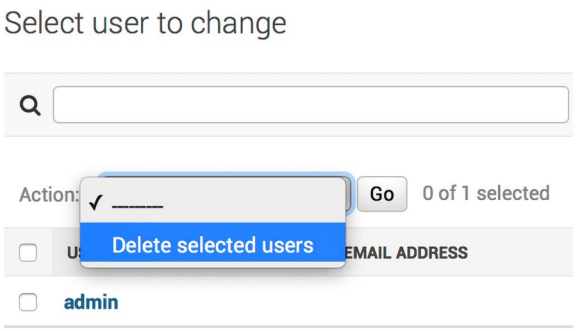


Figure 9.26: The drop-down menu for Django administration actions

You can create a custom action by writing a regular function that receives the following parameters:

- The current `ModelAdmin` being displayed
- The current request object as an `HttpRequest` instance
- A QuerySet for the objects selected by the user

This function will be executed when the action is triggered from the administration site.

You are going to create a custom administration action to download a list of orders as a CSV file.

Edit the `admin.py` file of the `orders` application and add the following code before the `OrderAdmin` class:

```
import csv
import datetime
from django.http import HttpResponse

def export_to_csv(modeladmin, request, queryset):
    opts = modeladmin.model._meta
    content_disposition = f'attachment; filename={opts.verbose_name}.csv'
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = content_disposition
    writer = csv.writer(response)
    fields = [field for field in opts.get_fields() if not \
              field.many_to_many and not field.one_to_many]
```

```
    # Write a first row with header information
    writer.writerow([field.verbose_name for field in fields])
    # Write data rows
    for obj in queryset:
        data_row = []
        for field in fields:
            value = getattr(obj, field.name)
            if isinstance(value, datetime.datetime):
                value = value.strftime('%d/%m/%Y')
            data_row.append(value)
        writer.writerow(data_row)
    return response
export_to_csv.short_description = 'Export to CSV'
```

In this code, you perform the following tasks:

1. You create an instance of `HttpResponse`, specifying the `text/csv` content type, to tell the browser that the response has to be treated as a CSV file. You also add a `Content-Disposition` header to indicate that the HTTP response contains an attached file.
2. You create a CSV `writer` object that will write to the `response` object.
3. You get the `model` fields dynamically using the `get_fields()` method of the model's `_meta` options. You exclude many-to-many and one-to-many relationships.
4. You write a header row including the field names.
5. You iterate over the given QuerySet and write a row for each object returned by the QuerySet. You take care of formatting `datetime` objects because the output value for CSV has to be a string.
6. You customize the display name for the action in the actions drop-down element of the administration site by setting a `short_description` attribute on the function.

You have created a generic administration action that can be added to any `ModelAdmin` class.

Finally, add the new `export_to_csv` administration action to the `OrderAdmin` class, as follows. New code is highlighted in bold:

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

Start the development server with the command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/orders/order/` in your browser. The resulting administration action should look like this:

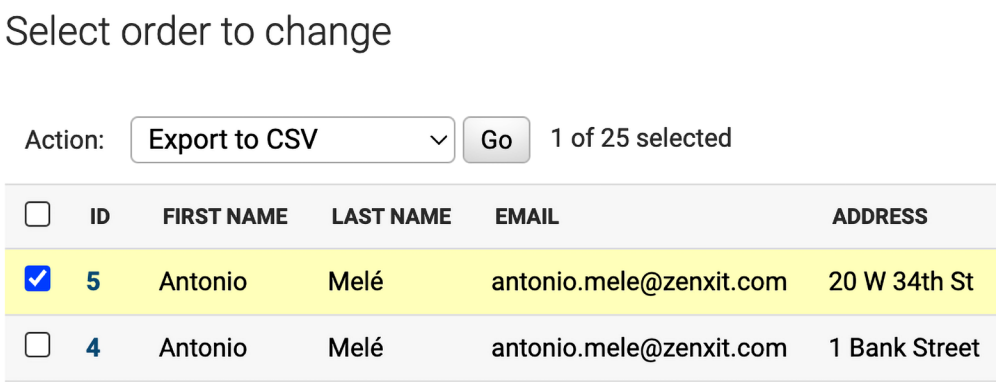


Figure 9.27: Using the custom Export to CSV administration action

Select some orders and choose the **Export to CSV** action from the select box, then click the **Go** button. Your browser will download the generated CSV file named `order.csv`. Open the downloaded file using a text editor. You should see content with the following format, including a header row and a row for each `Order` object you selected:

```
ID,first name,last name,email,address,postal
code,city,created,updated,paid,stripe id
5,Antonio,Melé,antonio.mele@zenxit.com,20 W 34th St,10001,New
York,24/03/2022,24/03/2022,True,pi_3KgzZVJ5UH88gi9T1l8ofnc6
...
```

As you can see, creating administration actions is pretty straightforward. You can learn more about generating CSV files with Django at `https://docs.djangoproject.com/en/4.1/howto/outputting-csv/`.

Next, you are going to customize the administration site further by creating a custom administration view.

Extending the administration site with custom views

Sometimes, you may want to customize the administration site beyond what is possible through configuring `ModelAdmin`, creating administration actions, and overriding administration templates. You might want to implement additional functionalities that are not available in existing administration views or templates. If this is the case, you need to create a custom administration view. With a custom view, you can build any functionality you want; you just have to make sure that only staff users can access your view and that you maintain the administration look and feel by making your template extend an administration template.

Let's create a custom view to display information about an order. Edit the `views.py` file of the `orders` application and add the following code highlighted in bold:

```
from django.urls import reverse
from django.shortcuts import render, redirect, get_object_or_404
from django.contrib.admin.views.decorators import staff_member_required
from .models import OrderItem, Order
from .forms import OrderCreateForm
, from .tasks import order_created
from cart.cart import Cart

def order_create(request):
    # ...

@staff_member_required
def admin_order_detail(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    return render(request,
                  'admin/orders/order/detail.html',
                  {'order': order})
```

The `staff_member_required` decorator checks that both the `is_active` and `is_staff` fields of the user requesting the page are set to `True`. In this view, you get the `Order` object with the given ID and render a template to display the order.

Next, edit the `urls.py` file of the `orders` application and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    path('create/', views.order_create, name='order_create'),
    path('admin/order/<int:order_id>/', views.admin_order_detail,
                                        name='admin_order_detail'),
]
```

Create the following file structure inside the `templates/` directory of the `orders` application:

```
admin/
    orders/
        order/
            detail.html
```

Edit the `detail.html` template and add the following content to it:

```
{% extends "admin/base_site.html" %}

{% block title %}
```

```
  Order {{ order.id }} {{ block.super }}
{% endblock %}

{% block breadcrumbs %}
  <div class="breadcrumbs">
    <a href="{% url "admin:index" %}">Home</a> &rsaquo;
    <a href="{% url "admin:orders_order_changelist" %}">Orders</a>
    &rsaquo;
    <a href="{% url "admin:orders_order_change" order.id %}">Order {{ order.id
}}</a>
    &rsaquo; Detail
  </div>
{% endblock %}

{% block content %}
<div class="module">
  <h1>Order {{ order.id }}</h1>
  <ul class="object-tools">
    <li>
      <a href="#" onclick="window.print();">
        Print order
      </a>
    </li>
  </ul>
  <table>
    <tr>
      <th>Created</th>
      <td>{{ order.created }}</td>
    </tr>
    <tr>
      <th>Customer</th>
      <td>{{ order.first_name }} {{ order.last_name }}</td>
    </tr>
    <tr>
      <th>E-mail</th>
      <td><a href="mailto:{{ order.email }}">{{ order.email }}</a></td>
    </tr>
    <tr>
      <th>Address</th>
    <td>
      {{ order.address }},
      {{ order.postal_code }} {{ order.city }}
    </td>
```

```
      </tr>
      <tr>
        <th>Total amount</th>
        <td>${{ order.get_total_cost }}</td>
      </tr>
      <tr>
        <th>Status</th>
        <td>{% if order.paid %}Paid{% else %}Pending payment{% endif %}</td>
      </tr>
      <tr>
        <th>Stripe payment</th>
        <td>
          {% if order.stripe_id %}
            <a href="{{ order.get_stripe_url }}" target="_blank">
              {{ order.stripe_id }}
            </a>
          {% endif %}
        </td>
      </tr>
    </table>
  </div>
  <div class="module">
    <h2>Items bought</h2>
    <table style="width:100%">
      <thead>
        <tr>
          <th>Product</th>
          <th>Price</th>
          <th>Quantity</th>
          <th>Total</th>
        </tr>
      </thead>
      <tbody>
        {% for item in order.items.all %}
          <tr class="row{% cycle "1" "2" %}">
            <td>{{ item.product.name }}</td>
            <td class="num">${{ item.price }}</td>
            <td class="num">{{ item.quantity }}</td>
            <td class="num">${{ item.get_cost }}</td>
          </tr>
        {% endfor %}
        <tr class="total">
          <td colspan="3">Total</td>
```

```
            <td class="num">${{ order.get_total_cost }}</td>
          </tr>
        </tbody>
      </table>
    </div>
    {% endblock %}
```

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

This is the template to display the details of an order on the administration site. This template extends the `admin/base_site.html` template of Django's administration site, which contains the main HTML structure and CSS styles. You use the blocks defined in the parent template to include your own content. You display information about the order and the items bought.

When you want to extend an administration template, you need to know its structure and identify existing blocks. You can find all administration templates at `https://github.com/django/django/tree/4.0/django/contrib/admin/templates/admin`.

You can also override an administration template if you need to. To do so, copy a template into your `templates/` directory, keeping the same relative path and filename. Django's administration site will use your custom template instead of the default one.

Finally, let's add a link to each `Order` object on the list display page of the administration site. Edit the `admin.py` file of the `orders` application and add the following code to it, above the `OrderAdmin` class:

```
from django.urls import reverse

def order_detail(obj):
    url = reverse('orders:admin_order_detail', args=[obj.id])
    return mark_safe(f'<a href="{url}">View</a>')
```

This is a function that takes an `Order` object as an argument and returns an HTML link for the `admin_order_detail` URL. Django escapes HTML output by default. You have to use the `mark_safe` function to avoid auto-escaping.

Then, edit the `OrderAdmin` class to display the link as follows. New code is highlighted in bold:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                    'address', 'postal_code', 'city', 'paid',
                    order_payment, 'created', 'updated',
                    order_detail]
    # ...
```

Start the development server with the command:

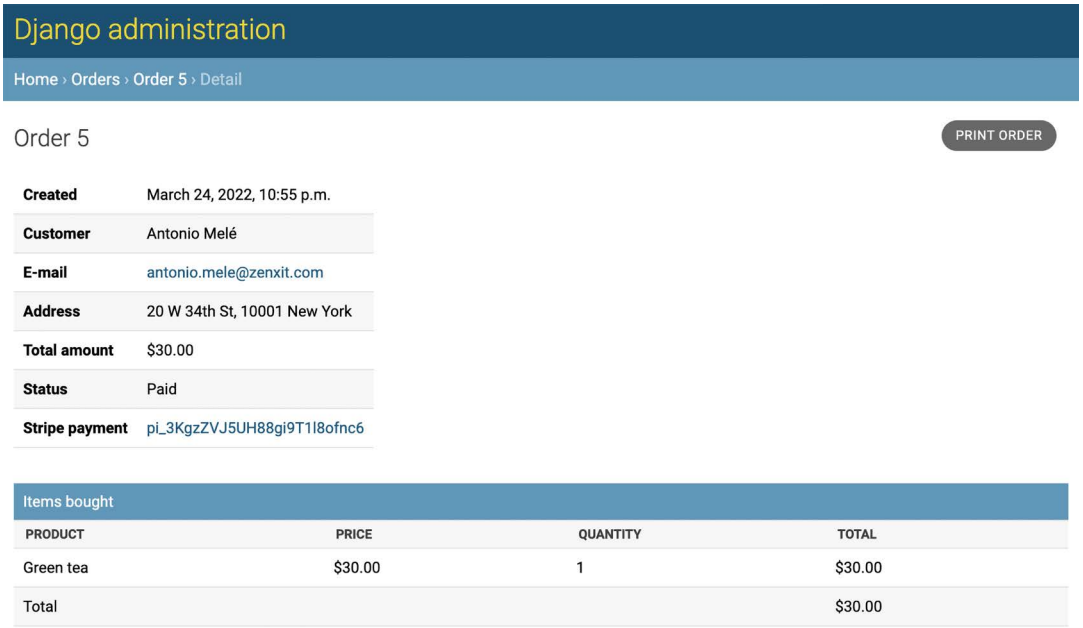
```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/orders/order/` in your browser. Each row includes a **View** link, as follows:

PAID	STRIPE PAYMENT	CREATED	UPDATED	ORDER DETAIL
✓	pi_3KgzZVJ5UH88gi9T1l8ofnc6	March 24, 2022, 10:55 p.m.	March 24, 2022, 7:44 p.m.	View

Figure 9.28: The View link included in each order row

Click on the **View** link for any order to load the custom order detail page. You should see a page like the following one:



Django administration

Home › Orders › Order 5 › Detail

Order 5 PRINT ORDER

Created	March 24, 2022, 10:55 p.m.
Customer	Antonio Melé
E-mail	antonio.mele@zenxit.com
Address	20 W 34th St, 10001 New York
Total amount	$30.00
Status	Paid
Stripe payment	pi_3KgzZVJ5UH88gi9T1l8ofnc6

Items bought

PRODUCT	PRICE	QUANTITY	TOTAL
Green tea	$30.00	1	$30.00
Total			$30.00

Figure 9.29: The custom order detail page on the administration site

Now that you have created the product detail page, you will learn how to generate order invoices in PDF format dynamically.

Generating PDF invoices dynamically

Now that you have a complete checkout and payment system, you can generate a PDF invoice for each order. There are several Python libraries to generate PDF files. One popular library to generate PDFs with Python code is ReportLab. You can find information about how to output PDF files with ReportLab at `https://docs.djangoproject.com/en/4.1/howto/outputting-pdf/`.

In most cases, you will have to add custom styles and formatting to your PDF files. You will find it more convenient to render an HTML template and convert it into a PDF file, keeping Python away from the presentation layer. You are going to follow this approach and use a module to generate PDF files with Django. You will use WeasyPrint, which is a Python library that can generate PDF files from HTML templates.

Installing WeasyPrint

First, install WeasyPrint's dependencies for your operating system from `https://doc.courtbouillon.org/weasyprint/stable/first_steps.html`. Then, install WeasyPrint via `pip` using the following command:

```
pip install WeasyPrint==56.1
```

Creating a PDF template

You need an HTML document as input for WeasyPrint. You are going to create an HTML template, render it using Django, and pass it to WeasyPrint to generate the PDF file.

Create a new template file inside the `templates/orders/order/` directory of the `orders` application and name it `pdf.html`. Add the following code to it:

```
<html>
<body>
  <h1>My Shop</h1>
  <p>
    Invoice no. {{ order.id }}<br>
    <span class="secondary">
      {{ order.created|date:"M d, Y" }}
    </span>
  </p>
  <h3>Bill to</h3>
  <p>
    {{ order.first_name }} {{ order.last_name }}<br>
    {{ order.email }}<br>
    {{ order.address }}<br>
    {{ order.postal_code }}, {{ order.city }}
  </p>
  <h3>Items bought</h3>
  <table>
    <thead>
      <tr>
        <th>Product</th>
        <th>Price</th>
        <th>Quantity</th>
        <th>Cost</th>
      </tr>
    </thead>
    <tbody>
      {% for item in order.items.all %}
```

```
        <tr class="row{% cycle "1" "2" %}">
          <td>{{ item.product.name }}</td>
          <td class="num">${{ item.price }}</td>
          <td class="num">{{ item.quantity }}</td>
          <td class="num">${{ item.get_cost }}</td>
        </tr>
      {% endfor %}
      <tr class="total">
        <td colspan="3">Total</td>
        <td class="num">${{ order.get_total_cost }}</td>
      </tr>
    </tbody>
  </table>

  <span class="{% if order.paid %}paid{% else %}pending{% endif %}">
    {% if order.paid %}Paid{% else %}Pending payment{% endif %}
  </span>
</body>
</html>
```

This is the template for the PDF invoice. In this template, you display all order details and an HTML `<table>` element including the products. You also include a message to display whether the order has been paid.

Rendering PDF files

You are going to create a view to generate PDF invoices for existing orders using the administration site. Edit the `views.py` file inside the `orders` application directory and add the following code to it:

```
from django.conf import settings
from django.http import HttpResponse
from django.template.loader import render_to_string
import weasyprint

@staff_member_required
def admin_order_pdf(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    html = render_to_string('orders/order/pdf.html',
                            {'order': order})
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = f'filename=order_{order.id}.pdf'
    weasyprint.HTML(string=html).write_pdf(response,
        stylesheets=[weasyprint.CSS(
```

```
            settings.STATIC_ROOT / 'css/pdf.css')])
    return response
```

This is the view to generate a PDF invoice for an order. You use the `staff_member_required` decorator to make sure only staff users can access this view.

You get the `Order` object with the given ID and you use the `render_to_string()` function provided by Django to render `orders/order/pdf.html`. The rendered HTML is saved in the `html` variable.

Then, you generate a new `HttpResponse` object specifying the `application/pdf` content type and including the `Content-Disposition` header to specify the filename. You use WeasyPrint to generate a PDF file from the rendered HTML code and write the file to the `HttpResponse` object.

You use the static file `css/pdf.css` to add CSS styles to the generated PDF file. Then, you load it from the local path by using the `STATIC_ROOT` setting. Finally, you return the generated response.

If you are missing the CSS styles, remember to copy the static files located in the `static/` directory of the `shop` application to the same location of your project.

You can find the contents of the directory at `https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter09/myshop/shop/static`.

Since you need to use the `STATIC_ROOT` setting, you have to add it to your project. This is the project's path where static files reside. Edit the `settings.py` file of the `myshop` project and add the following setting:

```
STATIC_ROOT = BASE_DIR / 'static'
```

Then, run the following command:

```
python manage.py collectstatic
```

You should see output that ends like this:

```
131 static files copied to 'code/myshop/static'.
```

The `collectstatic` command copies all static files from your applications into the directory defined in the `STATIC_ROOT` setting. This allows each application to provide its own static files using a `static/` directory containing them. You can also provide additional static file sources in the `STATICFILES_DIRS` setting. All of the directories specified in the `STATICFILES_DIRS` list will also be copied to the `STATIC_ROOT` directory when `collectstatic` is executed. Whenever you execute `collectstatic` again, you will be asked if you want to override the existing static files.

Edit the `urls.py` file inside the `orders` application directory and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    # ...
    path('admin/order/<int:order_id>/pdf/',
         views.admin_order_pdf,
```

```
        name='admin_order_pdf'),
]
```

Now you can edit the administration list display page for the `Order` model to add a link to the PDF file for each result. Edit the `admin.py` file inside the `orders` application and add the following code above the `OrderAdmin` class:

```
def order_pdf(obj):
    url = reverse('orders:admin_order_pdf', args=[obj.id])
    return mark_safe(f'<a href="{url}">PDF</a>')
order_pdf.short_description = 'Invoice'
```

If you specify a `short_description` attribute for your callable, Django will use it for the name of the column.

Add `order_pdf` to the `list_display` attribute of the `OrderAdmin` class, as follows:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                    'address', 'postal_code', 'city', 'paid',
                    order_payment, 'created', 'updated',
                    order_detail, order_pdf]
```

Make sure the development server is running. Open `http://127.0.0.1:8000/admin/orders/order/` in your browser. Each row should now include a **PDF** link, like this:

CREATED	UPDATED	ORDER DETAIL	INVOICE
March 24, 2022, 10:55 p.m.	March 24, 2022, 7:44 p.m.	View	PDF

Figure 9.30: The PDF link included in each order row

Click on the **PDF** link for any order. You should see a generated PDF file like the following one for orders that have not been paid yet:

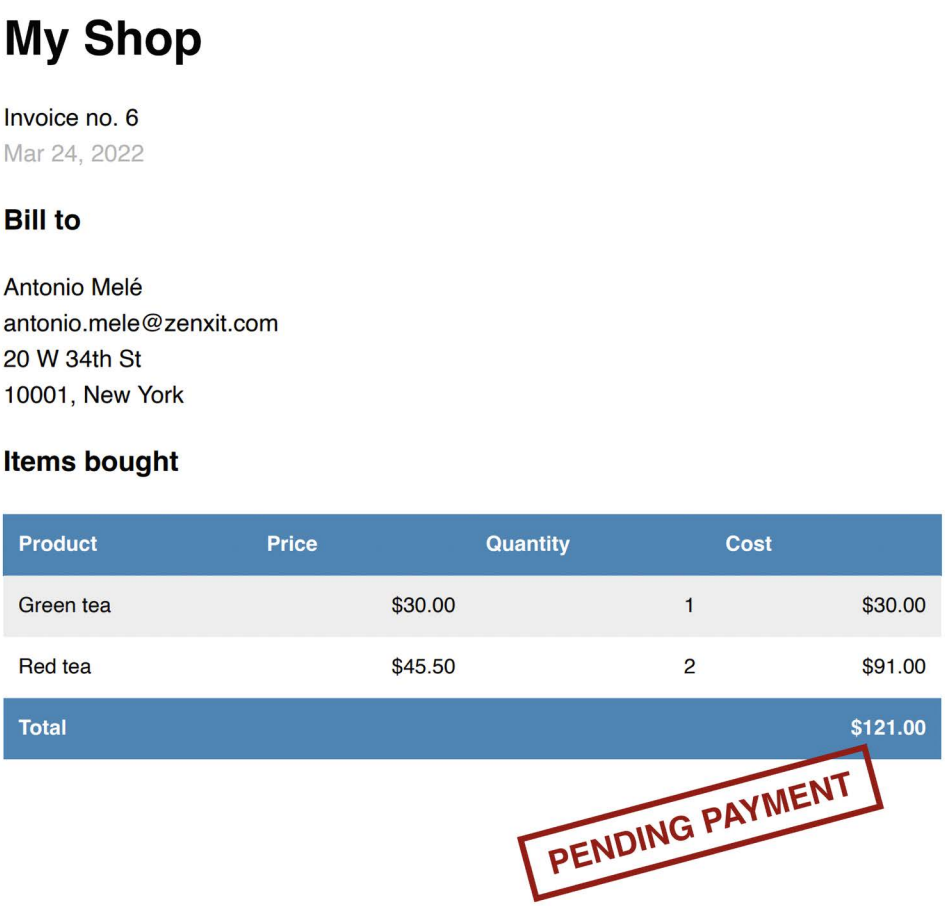


Figure 9.31: The PDF invoice for an unpaid order

For paid orders, you will see the following PDF file:

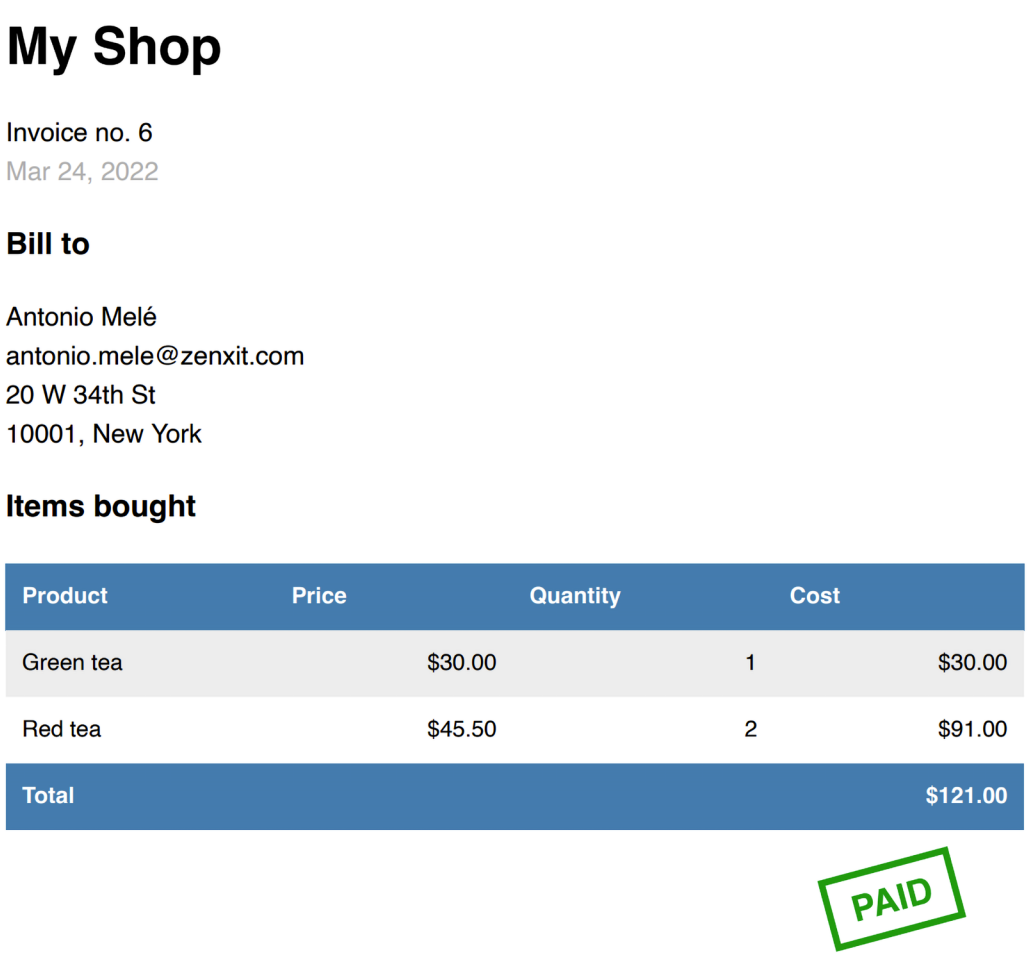


Figure 9.32: The PDF invoice for a paid order

Sending PDF files by email

When a payment is successful, you will send an automatic email to your customer including the generated PDF invoice. You will create an asynchronous task to perform this action.

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

```
from io import BytesIO
from celery import shared_task
import weasyprint
from django.template.loader import render_to_string
from django.core.mail import EmailMessage
```

```
from django.conf import settings
from orders.models import Order

@shared_task
def payment_completed(order_id):
    """
    Task to send an e-mail notification when an order is
    successfully paid.
    """
    order = Order.objects.get(id=order_id)
    # create invoice e-mail
    subject = f'My Shop - Invoice no. {order.id}'
    message = 'Please, find attached the invoice for your recent purchase.'
    email = EmailMessage(subject,
                         message,
                         'admin@myshop.com',
                         [order.email])
    # generate PDF
    html = render_to_string('orders/order/pdf.html', {'order': order})
    out = BytesIO()
    stylesheets=[weasyprint.CSS(settings.STATIC_ROOT / 'css/pdf.css')]
    weasyprint.HTML(string=html).write_pdf(out,
                                           stylesheets=stylesheets)
    # attach PDF file
    email.attach(f'order_{order.id}.pdf',
                 out.getvalue(),
                 'application/pdf')
    # send e-mail
    email.send()
```

You define the `payment_completed` task by using the `@shared_task` decorator. In this task, you use the `EmailMessage` class provided by Django to create an `email` object. Then, you render the template into the `html` variable. You generate the PDF file from the rendered template and output it to a `BytesIO` instance, which is an in-memory bytes buffer. Then, you attach the generated PDF file to the `EmailMessage` object using the `attach()` method, including the contents of the `out` buffer. Finally, you send the email.

Remember to set up your **Simple Mail Transfer Protocol** (**SMTP**) settings in the `settings.py` file of the project to send emails. You can refer to *Chapter 2*, *Enhancing Your Blog with Advanced Features*, to see a working example of an SMTP configuration. 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'
```

Let's add the `payment_completed` task to the webhook endpoint that handles payment completion events.

Edit the `webhooks.py` file of the `payment` application and modify it to make it look like this:

```
import stripe
from django.conf import settings
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from orders.models import Order
from .tasks import payment_completed

@csrf_exempt
def stripe_webhook(request):
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
    event = None

    try:
        event = stripe.Webhook.construct_event(
                    payload,
                    sig_header,
                    settings.STRIPE_WEBHOOK_SECRET)
    except ValueError as e:
        # Invalid payload
        return HttpResponse(status=400)
    except stripe.error.SignatureVerificationError as e:
        # Invalid signature
        return HttpResponse(status=400)

    if event.type == 'checkout.session.completed':
        session = event.data.object
        if session.mode == 'payment' and session.payment_status == 'paid':
            try:
                order = Order.objects.get(id=session.client_reference_id)
            except Order.DoesNotExist:
                return HttpResponse(status=404)
            # mark order as paid
            order.paid = True
            # store Stripe payment ID
            order.stripe_id = session.payment_intent
            order.save()
```

```
            # launch asynchronous task
            payment_completed.delay(order.id)

    return HttpResponse(status=200)
```

The `payment_completed` task is queued by calling its `delay()` method. The task will be added to the queue and will be executed asynchronously by a Celery worker as soon as possible.

Now you can complete a new checkout process in order to receive the PDF invoice in your email. If you are using the `console.EmailBackend` for your email backend, in the shell where you are running Celery you will be able to see the following output:

```
MIME-Version: 1.0
Subject: My Shop - Invoice no. 7
From: admin@myshop.com
To: antonio.mele@zenxit.com
Date: Sun, 27 Mar 2022 20:15:24 -0000
Message-ID: <164841212458.94972.10344068999595916799@antonios-mbp.home>

----------------8908668108717577350==
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Please, find attached the invoice for your recent purchase.
----------------8908668108717577350==
Content-Type: application/pdf
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="order_7.pdf"

JVBERi0xLjcKJfCflqQKMSAwIG9iago8PAovVHlwZSA...
```

This output shows that the email contains an attachment. You have learned how to attach files to emails and send them programmatically.

Congratulations! You have completed the Stripe integration and have added valuable functionality to your shop.

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-4-by-example/tree/main/Chapter09`

- Stripe website – `https://www.stripe.com/`
- Stripe Checkout documentation – `https://stripe.com/docs/payments/checkout`
- Creating a Stripe account – `https://dashboard.stripe.com/register`
- Stripe account settings – `https://dashboard.stripe.com/settings/account`
- Stripe Python library – `https://github.com/stripe/stripe-python`
- Stripe test API keys – `https://dashboard.stripe.com/test/apikeys`
- Stripe API keys documentation – `https://stripe.com/docs/keys`
- Stripe API version 2022-08-01 release notes – `https://stripe.com/docs/upgrades#2022-08-01`
- Stripe checkout session modes – `https://stripe.com/docs/api/checkout/sessions/object#checkout_session_object-mode`
- Building absolute URIs with Django – `https://docs.djangoproject.com/en/4.1/ref/request-response/#django.http.HttpRequest.build_absolute_uri`
- Creating Stripe sessions – `https://stripe.com/docs/api/checkout/sessions/create`
- Stripe-supported currencies – `https://stripe.com/docs/currencies`
- Stripe Payments dashboard – `https://dashboard.stripe.com/test/payments`
- Credit cards for testing payments with Stripe – `https://stripe.com/docs/testing`
- Stripe webhooks – `https://dashboard.stripe.com/test/webhooks`
- Types of events sent by Stripe – `https://stripe.com/docs/api/events/types`
- Installing the Stripe CLI – `https://stripe.com/docs/stripe-cli#install`
- Latest Stripe CLI release – `https://github.com/stripe/stripe-cli/releases/latest`
- Generating CSV files with Django – `https://docs.djangoproject.com/en/4.1/howto/outputting-csv/`
- Django administration templates – `https://github.com/django/django/tree/4.0/django/contrib/admin/templates/admin`
- Outputting PDF files with ReportLab – `https://docs.djangoproject.com/en/4.1/howto/outputting-pdf/`
- Installing WeasyPrint – `https://weasyprint.readthedocs.io/en/latest/install.html`
- Static files for this chapter – `https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter09/myshop/shop/static`

Summary

In this chapter, you integrated the Stripe payment gateway into your project and created a webhook endpoint to receive payment notifications. You built a custom administration action to export orders to CSV. You also customized the Django administration site using custom views and templates. Finally, you learned how to generate PDF files with WeasyPrint and how to attach them to emails.

The next chapter will teach you how to create a coupon system using Django sessions and you will build a product recommendation engine with Redis.

10

Extending Your Shop

In the previous chapter, you learned how to integrate a payment gateway into your shop. You also learned how to generate CSV and PDF files.

In this chapter, you will add a coupon system to your shop and create a product recommendation engine.

This chapter will cover the following points:

- Creating a coupon system
- Applying coupons to the shopping cart
- Applying coupons to orders
- Creating coupons for Stripe Checkout
- Storing products that are usually bought together
- Building a product recommendation engine with Redis

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter10`.

All the Python packages used in this chapter are included in the `requirements.txt` file in the source code for the chapter. You can follow the instructions to install each Python package in the following sections, or you can install all the requirements at once with the command `pip install -r requirements.txt`.

Creating a coupon system

Many online shops give out coupons to customers that can be redeemed for discounts on their purchases. An online coupon usually consists of a code that is given to users and is valid for a specific time frame.

You are going to create a coupon system for your shop. Your coupons will be valid for customers during a certain time frame. The coupons will not have any limitations in terms of the number of times they can be redeemed, and they will be applied to the total value of the shopping cart.

For this functionality, you will need to create a model to store the coupon code, a valid time frame, and the discount to apply.

Create a new application inside the `myshop` project using the following command:

```
python manage.py startapp coupons
```

Edit the `settings.py` file of `myshop` and add the application to the `INSTALLED_APPS` setting, as follows:

```
INSTALLED_APPS = [
    # ...
    'coupons.apps.CouponsConfig',
]
```

The new application is now active in your Django project.

Building the coupon model

Let's start by creating the `Coupon` model. Edit the `models.py` file of the `coupons` application and add the following code to it:

```
from django.db import models
from django.core.validators import MinValueValidator, \
                                   MaxValueValidator

class Coupon(models.Model):
    code = models.CharField(max_length=50,
                            unique=True)
    valid_from = models.DateTimeField()
    valid_to = models.DateTimeField()
    discount = models.IntegerField(
                   validators=[MinValueValidator(0),
                               MaxValueValidator(100)],
                   help_text='Percentage value (0 to 100)')
    active = models.BooleanField()

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

This is the model that you are going to use to store coupons. The `Coupon` model contains the following fields:

- `code`: The code that users have to enter in order to apply the coupon to their purchase.
- `valid_from`: The datetime value that indicates when the coupon becomes valid.

- `valid_to`: The datetime value that indicates when the coupon becomes invalid.
- `discount`: The discount rate to apply (this is a percentage, so it takes values from `0` to `100`). You use validators for this field to limit the minimum and maximum accepted values.
- `active`: A Boolean that indicates whether the coupon is active.

Run the following command to generate the initial migration for the `coupons` application:

```
python manage.py makemigrations
```

The output should include the following lines:

```
Migrations for 'coupons':
  coupons/migrations/0001_initial.py
    - Create model Coupon
```

Then, execute the next command to apply migrations:

```
python manage.py migrate
```

You should see an output that includes the following line:

```
Applying coupons.0001_initial... OK
```

The migrations have now been applied to the database. Let's add the `Coupon` model to the administration site. Edit the `admin.py` file of the `coupons` application and add the following code to it:

```
from django.contrib import admin
from .models import Coupon

@admin.register(Coupon)
class CouponAdmin(admin.ModelAdmin):
    list_display = ['code', 'valid_from', 'valid_to',
                    'discount', 'active']
    list_filter = ['active', 'valid_from', 'valid_to']
    search_fields = ['code']
```

The `Coupon` model is now registered on the administration site. Ensure that your local server is running with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/coupons/coupon/add/` in your browser.

You should see the following form:

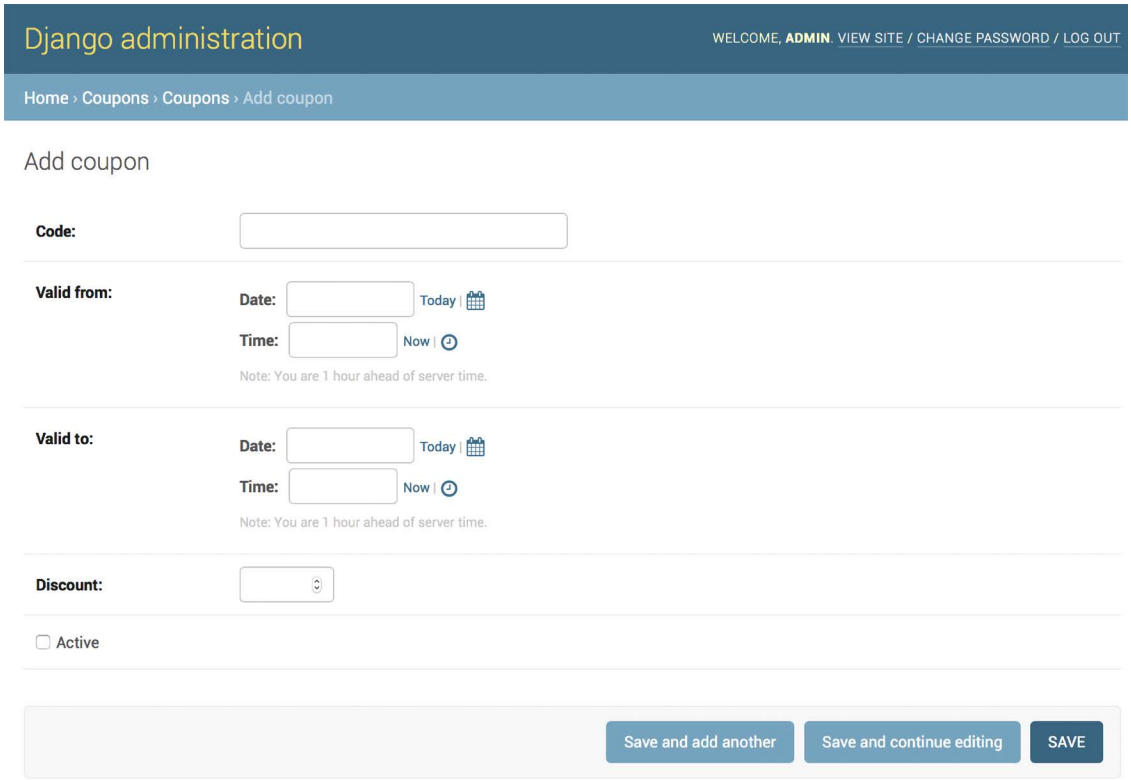


Figure 10.1: The Add coupon form on the Django administration site

Fill in the form to create a new coupon that is valid for the current date, make sure that you check the **Active** checkbox, and click the **SAVE** button. *Figure 10.2* shows an example of creating a coupon:

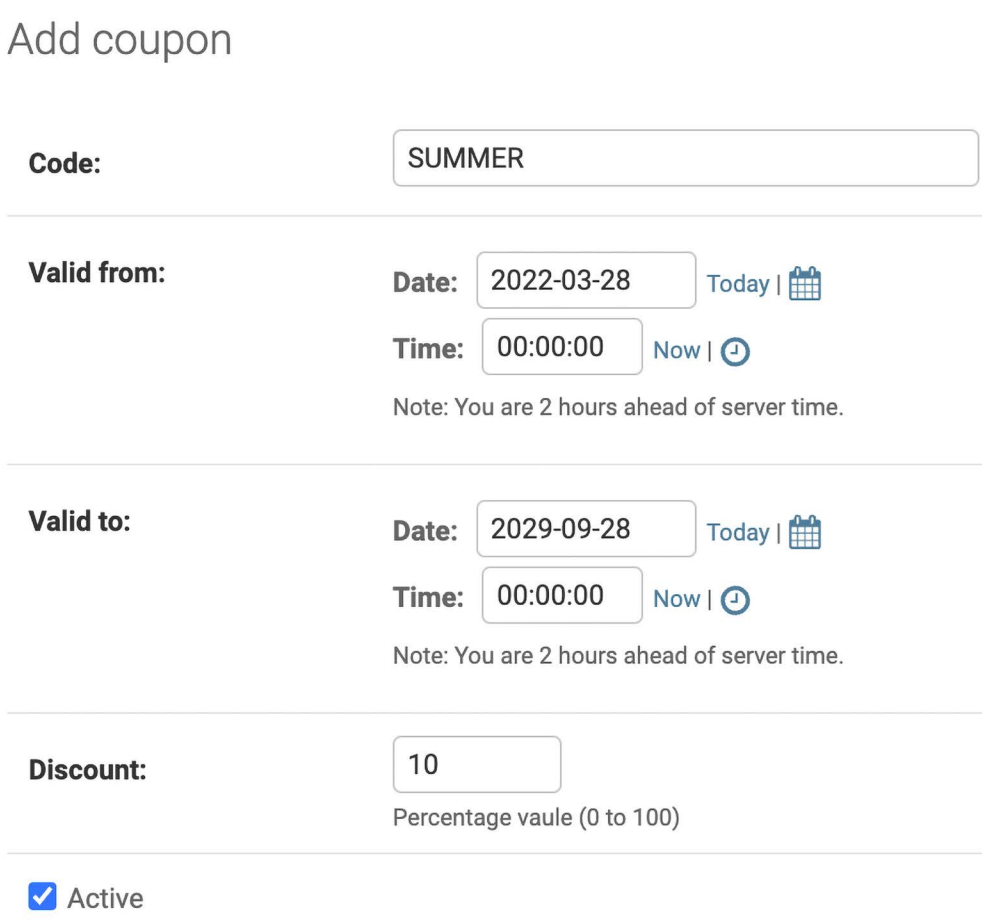


Figure 10.2: The Add coupon form with sample data

After creating the coupon, the coupon change list page on the administration site will look similar to *Figure 10.3*:

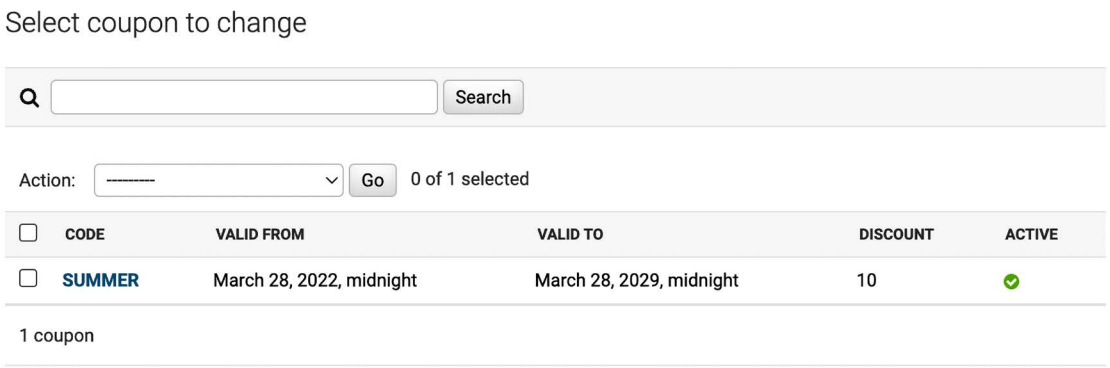


Figure 10.3: The coupon change list page on the Django administration site

Next, we will implement the functionality to apply coupons to the shopping cart.

Applying a coupon to the shopping cart

You can store new coupons and make queries to retrieve existing coupons. Now you need a way for customers to apply coupons to their purchases. The functionality to apply a coupon would be as follows:

1. The user adds products to the shopping cart.
2. The user can enter a coupon code in a form displayed on the shopping cart details page.
3. When the user enters a coupon code and submits the form, you look for an existing coupon with the given code that is currently valid. You have to check that the coupon code matches the one entered by the user, that the `active` attribute is `True`, and that the current datetime is between the `valid_from` and `valid_to` values.
4. If a coupon is found, you save it in the user's session and display the cart, including the discount applied to it and the updated total amount.
5. When the user places an order, you save the coupon to the given order.

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

```
from django import forms

class CouponApplyForm(forms.Form):
    code = forms.CharField()
```

This is the form that you are going to use for the user to enter a coupon code. Edit the `views.py` file inside the `coupons` application and add the following code to it:

```
from django.shortcuts import render, redirect
from django.utils import timezone
from django.views.decorators.http import require_POST
from .models import Coupon
from .forms import CouponApplyForm

@require_POST
def coupon_apply(request):
    now = timezone.now()
    form = CouponApplyForm(request.POST)
    if form.is_valid():
        code = form.cleaned_data['code']
        try:
            coupon = Coupon.objects.get(code__iexact=code,
                                        valid_from__lte=now,
                                        valid_to__gte=now,
                                        active=True)
            request.session['coupon_id'] = coupon.id
        except Coupon.DoesNotExist:
            request.session['coupon_id'] = None
    return redirect('cart:cart_detail')
```

The `coupon_apply` view validates the coupon and stores it in the user's session. You apply the `require_POST` decorator to this view to restrict it to `POST` requests. In the view, you perform the following tasks:

1. You instantiate the `CouponApplyForm` form using the posted data and check that the form is valid.
2. If the form is valid, you get the `code` entered by the user from the form's `cleaned_data` dictionary. You try to retrieve the `Coupon` object with the given code. You use the `iexact` field lookup to perform a case-insensitive exact match. The coupon has to be currently active (`active=True`) and valid for the current datetime. You use Django's `timezone.now()` function to get the current timezone-aware datetime, and you compare it with the `valid_from` and `valid_to` fields by performing the `lte` (less than or equal to) and `gte` (greater than or equal to) field lookups, respectively.
3. You store the coupon ID in the user's session.
4. You redirect the user to the `cart_detail` URL to display the cart with the coupon applied.

You need a URL pattern for the `coupon_apply` view. Create a new file inside the `coupons` application directory and name it `urls.py`. Add the following code to it:

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

app_name = 'coupons'

urlpatterns = [
    path('apply/', views.coupon_apply, name='apply'),
]
```

Then, edit the main `urls.py` of the `myshop` project and include the `coupons` URL patterns with the following line highlighted in bold:

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

Remember to place this pattern before the `shop.urls` pattern.

Now, edit the `cart.py` file of the `cart` application. Include the following import:

```
from coupons.models import Coupon
```

Add the following code highlighted in bold to the end of the `__init__()` method of the `Cart` class to initialize the coupon from the current session:

```
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
        # store current applied coupon
        self.coupon_id = self.session.get('coupon_id')
```

In this code, you try to get the `coupon_id` session key from the current session and store its value in the `Cart` object. Add the following methods highlighted in bold to the `Cart` object:

```
class Cart:
    # ...

    @property
    def coupon(self):
        if self.coupon_id:
            try:
                return Coupon.objects.get(id=self.coupon_id)
            except Coupon.DoesNotExist:
                pass
        return None

    def get_discount(self):
        if self.coupon:
            return (self.coupon.discount / Decimal(100)) \
                * self.get_total_price()
        return Decimal(0)

    def get_total_price_after_discount(self):
        return self.get_total_price() - self.get_discount()
```

These methods are as follows:

- `coupon()`: You define this method as a `property`. If the cart contains a `coupon_id` attribute, the `Coupon` object with the given ID is returned.
- `get_discount()`: If the cart contains a coupon, you retrieve its discount rate and return the amount to be deducted from the total amount of the cart.
- `get_total_price_after_discount()`: You return the total amount of the cart after deducting the amount returned by the `get_discount()` method.

The `Cart` class is now prepared to handle a coupon applied to the current session and apply the corresponding discount.

Let's include the coupon system in the cart's detail view. Edit the `views.py` file of the `cart` application and add the following import to the top of the file:

```
from coupons.forms import CouponApplyForm
```

Further down, edit the `cart_detail` view and add the new form to it, as follows:

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
```

```
        item['update_quantity_form'] = CartAddProductForm(initial={
                                'quantity': item['quantity'],
                                'override': True})
    coupon_apply_form = CouponApplyForm()
    return render(request,
                  'cart/detail.html',
                  {'cart': cart,
                   'coupon_apply_form': coupon_apply_form})
```

Edit the `cart/detail.html` template of the `cart` application and locate the following lines:

```
<tr class="total">
  <td>Total</td>
  <td colspan="4"></td>
  <td class="num">${{ cart.get_total_price }}</td>
</tr>
```

Replace them with the following code:

```
{% if cart.coupon %}
  <tr class="subtotal">
    <td>Subtotal</td>
    <td colspan="4"></td>
    <td class="num">${{ cart.get_total_price|floatformat:2 }}</td>
  </tr>
  <tr>
    <td>
      "{{ cart.coupon.code }}" coupon
      ({{ cart.coupon.discount }}% off)
    </td>
    <td colspan="4"></td>
    <td class="num neg">
      - ${{ cart.get_discount|floatformat:2 }}
    </td>
  </tr>
{% endif %}
<tr class="total">
  <td>Total</td>
  <td colspan="4"></td>
  <td class="num">
    ${{ cart.get_total_price_after_discount|floatformat:2 }}
  </td>
</tr>
```

This is the code for displaying an optional coupon and its discount rate. If the cart contains a coupon, you display the first row, including the total amount of the cart as the subtotal. Then, you use a second row to display the current coupon applied to the cart. Finally, you display the total price, including any discount, by calling the `get_total_price_after_discount()` method of the `cart` object.

In the same file, include the following code after the `</table>` HTML tag:

```
<p>Apply a coupon:</p>
<form action="{% url "coupons:apply" %}" method="post">
  {{ coupon_apply_form }}
  <input type="submit" value="Apply">
  {% csrf_token %}
</form>
```

This will display the form to enter a coupon code and apply it to the current cart.

Open `http://127.0.0.1:8000/` in your browser and add a product to the cart. You will see that the shopping cart page now includes a form to apply a coupon:

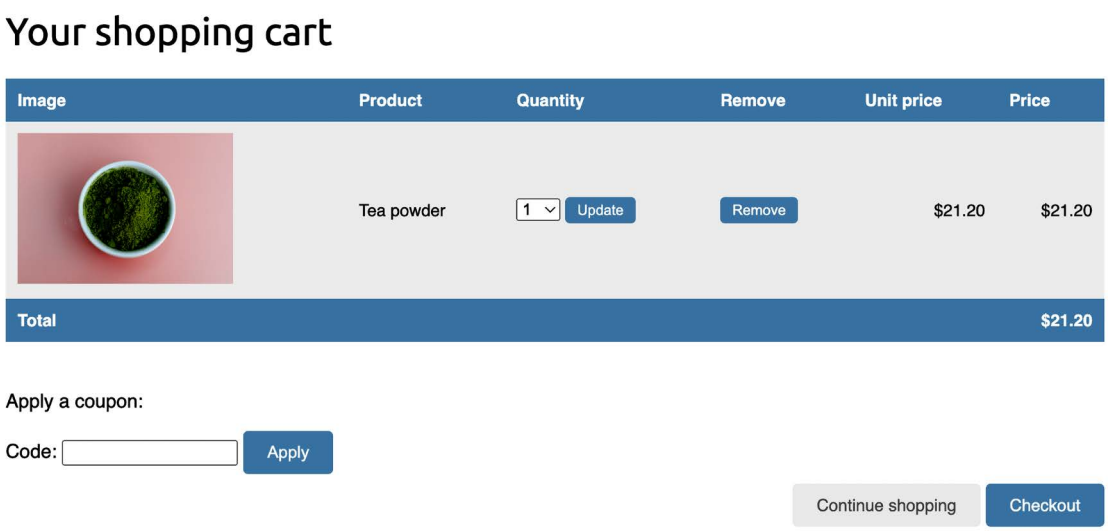


Figure 10.4: The cart detail page, including a form to apply a coupon

Image of *Tea powder*: Photo by Phuong Nguyen on Unsplash

In the **Code** field, enter the coupon code you created using the administration site:

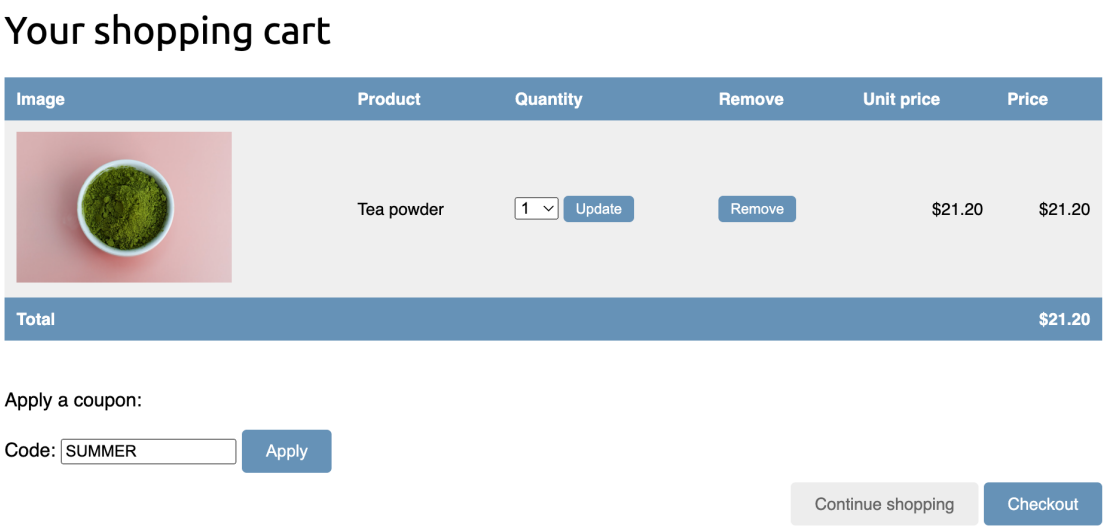


Figure 10.5: The cart detail page, including a coupon code on the form

Click the **Apply** button. The coupon will be applied, and the cart will display the coupon discount as follows:

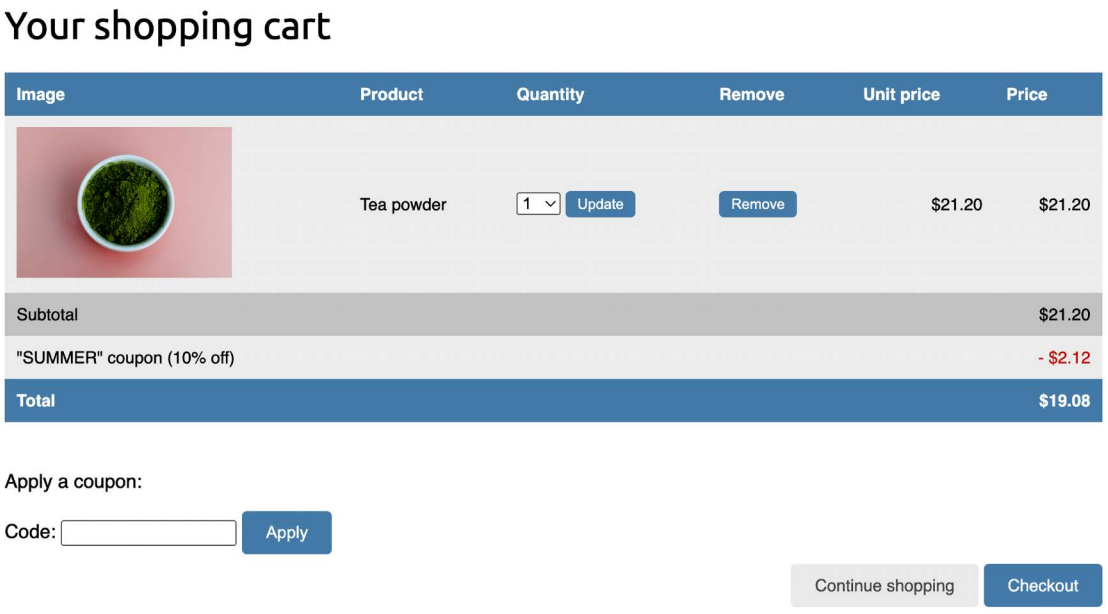


Figure 10.6: The cart detail page, including the coupon applied

Let's add the coupon to the next step of the purchase process. Edit the `orders/order/create.html` template of the `orders` application and locate the following lines:

```
<ul>
  {% for item in cart %}
    <li>
      {{ item.quantity }}x {{ item.product.name }}
      <span>${{ item.total_price }}</span>
    </li>
  {% endfor %}
</ul>
```

Replace them with the following code:

```
<ul>
  {% for item in cart %}
    <li>
      {{ item.quantity }}x {{ item.product.name }}
      <span>${{ item.total_price|floatformat:2 }}</span>
    </li>
  {% endfor %}
  {% if cart.coupon %}
    <li>
      "{{ cart.coupon.code }}" ({{ cart.coupon.discount }}% off)
      <span class="neg">- ${{ cart.get_discount|floatformat:2 }}</span>
    </li>
  {% endif %}
</ul>
```

The order summary should now include the coupon applied, if there is one. Now find the following line:

```
<p>Total: ${{ cart.get_total_price }}</p>
```

Replace it with the following:

```
<p>Total: ${{ cart.get_total_price_after_discount|floatformat:2 }}</p>
```

By doing this, the total price will also be calculated by applying the discount of the coupon.

Open `http://127.0.0.1:8000/orders/create/` in your browser. You should see that the order summary includes the applied coupon, as follows:

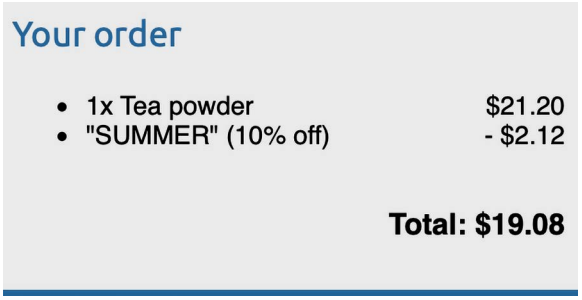


Figure 10.7: The order summary, including the coupon applied to the cart

Users can now apply coupons to their shopping cart. However, you still need to store coupon information in the order that it is created when users check out the cart.

Applying coupons to orders

You are going to store the coupon that was applied to each order. First, you need to modify the `Order` model to store the related `Coupon` object, if there is one.

Edit the `models.py` file of the `orders` application and add the following imports to it:

```
from decimal import Decimal
from django.core.validators import MinValueValidator, \
                                   MaxValueValidator
from coupons.models import Coupon
```

Then, add the following fields to the `Order` model:

```
class Order(models.Model):
    # ...
    coupon = models.ForeignKey(Coupon,
                               related_name='orders',
                               null=True,
                               blank=True,
                               on_delete=models.SET_NULL)
    discount = models.IntegerField(default=0,
                                   validators=[MinValueValidator(0),
                                       MaxValueValidator(100)])
```

These fields allow you to store an optional coupon for the order and the discount percentage applied with the coupon. The discount is stored in the related `Coupon` object, but you can include it in the `Order` model to preserve it if the coupon has been modified or deleted. You set `on_delete` to `models.SET_NULL` so that if the coupon gets deleted, the `coupon` field is set to `Null`, but the discount is preserved.

You need to create a migration to include the new fields of the `Order` model. Run the following command from the command line:

```
python manage.py makemigrations
```

You should see an output like the following:

```
Migrations for 'orders':
  orders/migrations/0003_order_coupon_order_discount.py
    - Add field coupon to order
    - Add field discount to order
```

Apply the new migration with the following command:

```
python manage.py migrate orders
```

You should see the following confirmation indicating that the new migration has been applied:

```
Applying orders.0003_order_coupon_order_discount... OK
```

The `Order` model field changes are now synced with the database.

Edit the `models.py` file, and add two new methods, `get_total_cost_before_discount()` and `get_discount()`, to the `Order` model like this. The new code is highlighted in bold:

```
class Order(models.Model):
    # ...
    def get_total_cost_before_discount(self):
        return sum(item.get_cost() for item in self.items.all())

    def get_discount(self):
        total_cost = self.get_total_cost_before_discount()
        if self.discount:
            return total_cost * (self.discount / Decimal(100))
        return Decimal(0)
```

Then, edit the `get_total_cost()` method of the `Order` model as follows. The new code is highlighted in bold:

```
    def get_total_cost(self):
        total_cost = self.get_total_cost_before_discount()
        return total_cost - self.get_discount()
```

The `get_total_cost()` method of the `Order` model will now take into account the discount applied, if there is one.

Edit the `views.py` file of the `orders` application and modify the `order_create` view to save the related coupon and its discount when creating a new order. Add the following code highlighted in bold to the `order_create` view:

```
def order_create(request):
    cart = Cart(request)
    if request.method == 'POST':
        form = OrderCreateForm(request.POST)
        if form.is_valid():
            order = form.save(commit=False)
            if cart.coupon:
                order.coupon = cart.coupon
                order.discount = cart.coupon.discount
            order.save()
            for item in cart:
                OrderItem.objects.create(order=order,
                                         product=item['product'],
                                         price=item['price'],
                                         quantity=item['quantity'])
            # clear the cart
            cart.clear()
            # launch asynchronous task
            order_created.delay(order.id)
            # set the order in the session
            request.session['order_id'] = order.id
            # redirect for payment
            return redirect(reverse('payment:process'))
    else:
        form = OrderCreateForm()
    return render(request,
                  'orders/order/create.html',
                  {'cart': cart, 'form': form})
```

In the new code, you create an `Order` object using the `save()` method of the `OrderCreateForm` form. You avoid saving it to the database yet by using `commit=False`. If the cart contains a coupon, you store the related coupon and the discount that was applied. Then, you save the `order` object to the database.

Edit the `payment/process.html` template of the `payment` application and locate the following lines:

```
<tr class="total">
  <td>Total</td>
  <td colspan="4"></td>
  <td class="num">${{ order.get_total_cost }}</td>
</tr>
```

Replace them with the following code. New lines are highlighted in bold:

```
{% if order.coupon %}
  <tr class="subtotal">
    <td>Subtotal</td>
    <td colspan="3"></td>
    <td class="num">
      ${{ order.get_total_cost_before_discount|floatformat:2 }}
    </td>
  </tr>
  <tr>
    <td>
      "{{ order.coupon.code }}" coupon
      ({{ order.discount }}% off)
    </td>
    <td colspan="3"></td>
    <td class="num neg">
      - ${{ order.get_discount|floatformat:2 }}
    </td>
  </tr>
{% endif %}
<tr class="total">
  <td>Total</td>
  <td colspan="3"></td>
  <td class="num">
    ${{ order.get_total_cost|floatformat:2 }}
  </td>
</tr>
```

We have updated the order summary before payment.

Make sure that the development server is running with the following command:

```
python manage.py runserver
```

Make sure Docker is running, and execute the following command in another shell to start the RabbitMQ server with Docker:

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

Open another shell and start the Celery worker from your project directory with the following command:

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

Open an additional shell and execute the following command to forward Stripe events to your local webhook URL:

```
stripe listen --forward-to localhost:8000/payment/webhook/
```

Open `http://127.0.0.1:8000/` in your browser and create an order using the coupon you created. After validating the items in the shopping cart, on the **Order summary** page, you will see the coupon applied to the order:

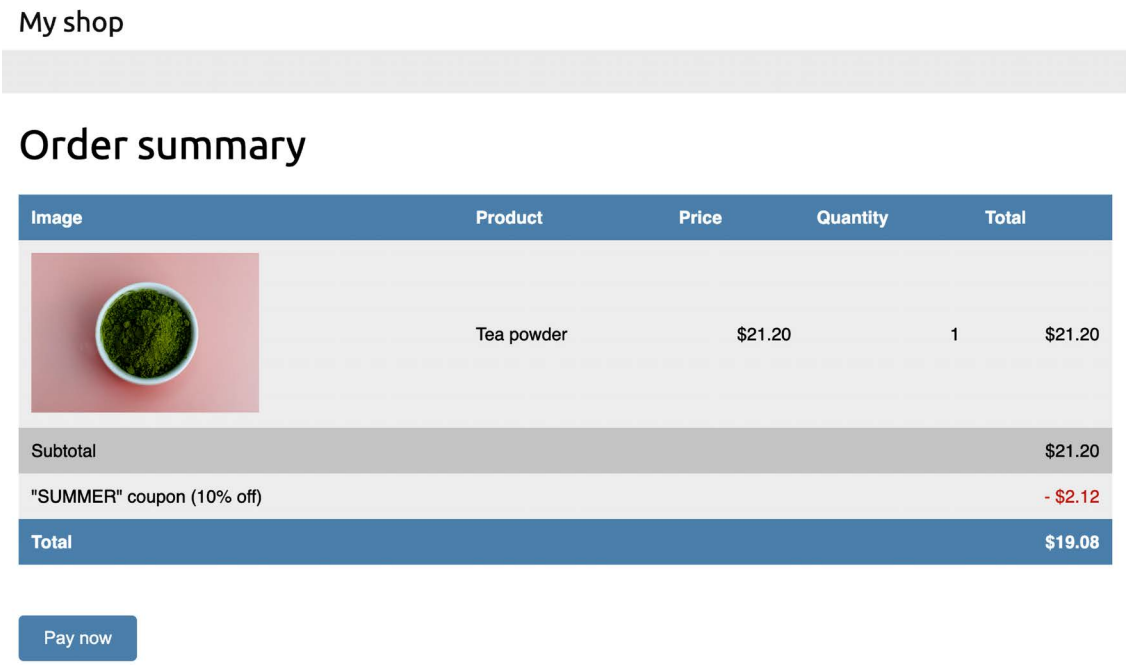


Figure 10.8: The Order summary page, including the coupon applied to the order

If you click on **Pay now**, you will see that Stripe is not aware of the discount applied, as displayed in *Figure 10.9*:

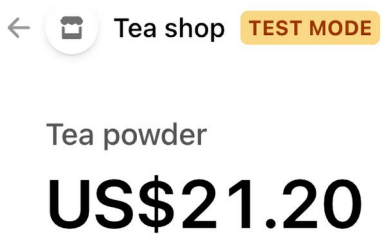


Figure 10.9: The item details of the Stripe Checkout page, including no discount coupon

Stripe shows the full amount to be paid without any deduction. This is because we are not passing on the discount to Stripe. Remember that in the `payment_process` view, we pass the order items as `line_items` to Stripe, including the cost and quantity of each order item.

Creating coupons for Stripe Checkout

Stripe allows you to define discount coupons and link them to one-time payments. You can find more information about creating discounts for Stripe Checkout at `https://stripe.com/docs/payments/checkout/discounts`.

Let's edit the `payment_process` view to create a coupon for Stripe Checkout. Edit the `views.py` file of the `payment` application and add the following code highlighted in bold to the `payment_process` view:

```
def payment_process(request):
    order_id = request.session.get('order_id', None)
    order = get_object_or_404(Order, id=order_id)

    if request.method == 'POST':
        success_url = request.build_absolute_uri(
                        reverse('payment:completed'))
        cancel_url = request.build_absolute_uri(
                        reverse('payment:canceled'))

        # Stripe checkout session data
        session_data = {
            'mode': 'payment',
            'client_reference_id': order.id,
            'success_url': success_url,
            'cancel_url': cancel_url,
            'line_items': []
        }
        # add order items to the Stripe checkout session
        for item in order.items.all():
            session_data['line_items'].append({
                'price_data': {
                    'unit_amount': int(item.price * Decimal('100')),
                    'currency': 'usd',
                    'product_data': {
                        'name': item.product.name,
                    },
                },
                'quantity': item.quantity,
            })

        # Stripe coupon
        if order.coupon:
```

```
            stripe_coupon = stripe.Coupon.create(
                                    name=order.coupon.code,
                                    percent_off=order.discount,
                                    duration='once')
            session_data['discounts'] = [{
                'coupon': stripe_coupon.id
            }]

        # create Stripe checkout session
        session = stripe.checkout.Session.create(**session_data)

        # redirect to Stripe payment form
        return redirect(session.url, code=303)

    else:
        return render(request, 'payment/process.html', locals())
```

In the new code, you check if the order has a related coupon. In that case, you use the Stripe SDK to create a Stripe coupon using `stripe.Coupon.create()`. You use the following attributes for the coupon:

- `name`: The `code` of the coupon related to the `order` object is used.
- `percent_off`: The `discount` of the `order` object is issued.
- `duration`: The value `once` is used. This indicates to Stripe that this is a coupon for a one-time payment.

After creating the coupon, its `id` is added to the `session_data` dictionary used to create the Stripe Checkout session. This links the coupon to the checkout session.

Open `http://127.0.0.1:8000/` in your browser and complete a purchase using the coupon you created. When redirected to the Stripe Checkout page, you will see the coupon applied:

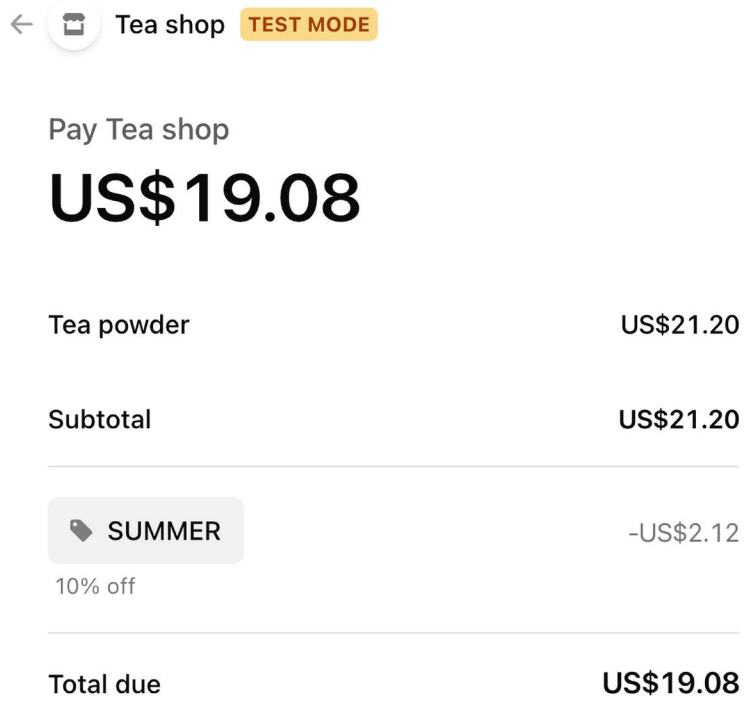


Figure 10.10: The item details of the Stripe Checkout page, including a discount coupon named SUMMER

The Stripe Checkout page now includes the order coupon, and the total amount to pay now includes the amount deducted using the coupon.

Complete the purchase and then open `http://127.0.0.1:8000/admin/orders/order/` in your browser. Click on the `order` object for which the coupon was used. The edit form will display the discount applied, as shown in *Figure 10.11*:

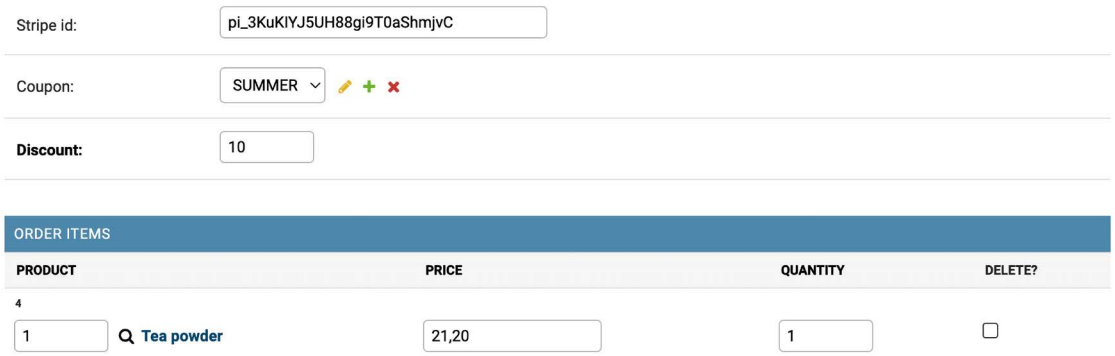


Figure 10.11: The order edit form, including the coupon and discount applied

You are successfully storing coupons for orders and processing payments with discounts. Next, you will add coupons to the order detail view of the administration site and to PDF invoices for orders.

Adding coupons to orders on the administration site and to PDF invoices

Let's add the coupon to the order detail page on the administration site. Edit the `admin/orders/order/detail.html` template of the `orders` application and add the following code highlighted in bold:

```
...
<table style="width:100%">
  ...
  <tbody>
    {% for item in order.items.all %}
      <tr class="row{% cycle "1" "2" %}">
        <td>{{ item.product.name }}</td>
        <td class="num">${{ item.price }}</td>
        <td class="num">{{ item.quantity }}</td>
        <td class="num">${{ item.get_cost }}</td>
      </tr>
    {% endfor %}

    {% if order.coupon %}
      <tr class="subtotal">
        <td colspan="3">Subtotal</td>
        <td class="num">
          ${{ order.get_total_cost_before_discount|floatformat:2 }}
```

```
        </td>
      </tr>
      <tr>
        <td colspan="3">
          "{{ order.coupon.code }}" coupon
          ({{ order.discount }}% off)
        </td>
        <td class="num neg">
          - ${{ order.get_discount|floatformat:2 }}
        </td>
      </tr>
    {% endif %}

    <tr class="total">
      <td colspan="3">Total</td>
      <td class="num">
        ${{ order.get_total_cost|floatformat:2 }}
      </td>
    </tr>
  </tbody>
</table>
...
```

Access `http://127.0.0.1:8000/admin/orders/order/` with your browser, and click on the **View** link of the latest order. The **Items bought** table will now include the coupon used, as shown in *Figure 10.12*:

Items bought			
PRODUCT	**PRICE**	**QUANTITY**	**TOTAL**
Tea powder	$21.20	2	$42.40
Subtotal			$42.40
"SUMMER" coupon (10% off)			- $4.24
Total			$38.16

Figure 10.12: The product detail page on the administration site, including the coupon used

Now, let's modify the order invoice template to include the coupon used for the order. Edit the `orders/order/detail.pdf` template of the `orders` application and add the following code highlighted in bold:

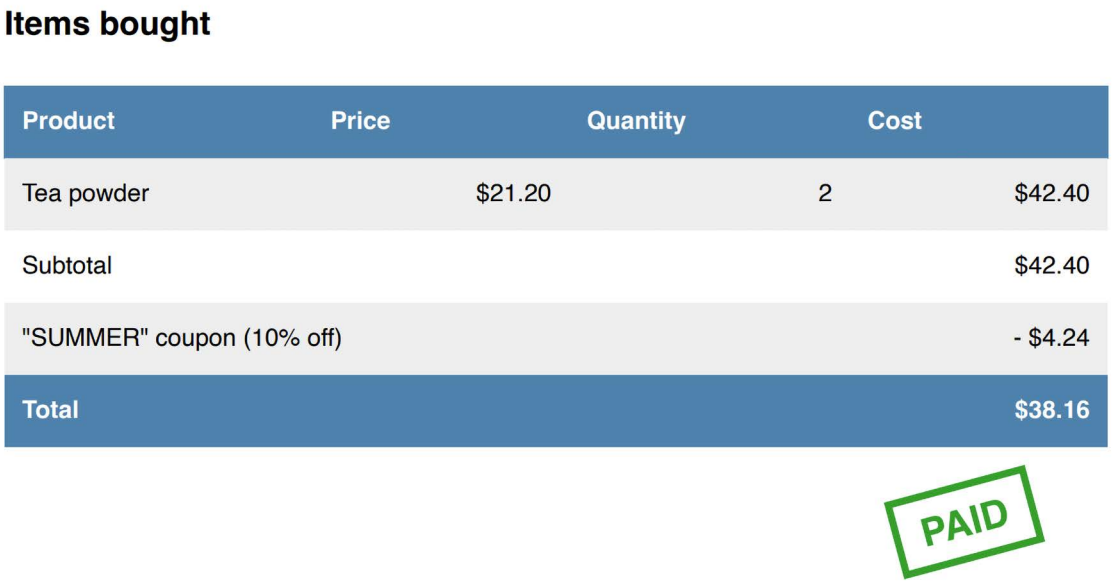
```
...
<table>
  <thead>
    <tr>
```

```
      <th>Product</th>
      <th>Price</th>
      <th>Quantity</th>
      <th>Cost</th>
    </tr>
  </thead>
  <tbody>
    {% for item in order.items.all %}
      <tr class="row{% cycle "1" "2" %}">
        <td>{{ item.product.name }}</td>
        <td class="num">${{ item.price }}</td>
        <td class="num">{{ item.quantity }}</td>
        <td class="num">${{ item.get_cost }}</td>
      </tr>
    {% endfor %}

    {% if order.coupon %}
      <tr class="subtotal">
        <td colspan="3">Subtotal</td>
        <td class="num">
          ${{ order.get_total_cost_before_discount|floatformat:2 }}
        </td>
      </tr>
      <tr>
        <td colspan="3">
          "{{ order.coupon.code }}" coupon
          ({{ order.discount }}% off)
        </td>
        <td class="num neg">
          - ${{ order.get_discount|floatformat:2 }}
        </td>
      </tr>
    {% endif %}

    <tr class="total">
      <td colspan="3">Total</td>
      <td class="num">${{ order.get_total_cost|floatformat:2 }}</td>
    </tr>
  </tbody>
</table>
...
```

Access `http://127.0.0.1:8000/admin/orders/order/` with your browser, and click on the **PDF** link of the latest order. The **Items bought** table will now include the coupon used, as shown in *Figure 10.13*:



Items bought

Product	Price	Quantity	Cost
Tea powder	$21.20	2	$42.40
Subtotal			$42.40
"SUMMER" coupon (10% off)			- $4.24
Total			**$38.16**

PAID

Figure 10.13: The PDF order invoice, including the coupon used

You successfully added a coupon system to your shop. Next, you are going to build a product recommendation engine.

Building a recommendation engine

A recommendation engine is a system that predicts the preference or rating that a user would give to an item. The system selects relevant items for a user based on their behavior and the knowledge it has about them. Nowadays, recommendation systems are used in many online services. They help users by selecting the stuff they might be interested in from the vast amount of available data that is irrelevant to them. Offering good recommendations enhances user engagement. E-commerce sites also benefit from offering relevant product recommendations by increasing their average revenue per user.

You are going to create a simple, yet powerful, recommendation engine that suggests products that are usually bought together. You will suggest products based on historical sales, thus identifying products that are usually bought together. You are going to suggest complementary products in two different scenarios:

- **Product detail page**: You will display a list of products that are usually bought with the given product. This will be displayed as *users who bought this also bought X, Y, and Z*. You need a data structure that allows you to store the number of times each product has been bought together with the product being displayed.

- **Cart detail page:** Based on the products that users add to the cart, you are going to suggest products that are usually bought together with these ones. In this case, the score you calculate to obtain related products has to be aggregated.

You are going to use Redis to store products that are usually purchased together. Remember that you already used Redis in *Chapter 7*, *Tracking User Actions*. If you haven't installed Redis yet, you can find installation instructions in that chapter.

Recommending products based on previous purchases

We will recommend products to users based on the items that are frequently bought together. For that, we are going to store a key in Redis for each product bought on the site. The product key will contain a Redis sorted set with scores. Every time a new purchase is completed, we will increment the score by 1 for each product bought together. The sorted set will allow you to give scores to products that are bought together. We will use the number of times the product is bought with another product as the score for that item.

Remember to install `redis-py` in your environment using the following command:

```
pip install redis==4.3.4
```

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

```
# Redis settings
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 1
```

These are the settings required to establish a connection with the Redis server. Create a new file inside the `shop` application directory and name it `recommender.py`. Add the following code to it:

```
import redis
from django.conf import settings
from .models import Product

# connect to redis
r = redis.Redis(host=settings.REDIS_HOST,
                port=settings.REDIS_PORT,
                db=settings.REDIS_DB)

class Recommender:
    def get_product_key(self, id):
        return f'product:{id}:purchased_with'

    def products_bought(self, products):
        product_ids = [p.id for p in products]
```

```
        for product_id in product_ids:
            for with_id in product_ids:
                # get the other products bought with each product
                if product_id != with_id:
                    # increment score for product purchased together
                    r.zincrby(self.get_product_key(product_id),
                              1,
                              with_id)
```

This is the `Recommender` class, which will allow you to store product purchases and retrieve product suggestions for a given product or products.

The `get_product_key()` method receives an ID of a `Product` object and builds the Redis key for the sorted set where related products are stored, which looks like `product:[id]:purchased_with`.

The `products_bought()` method receives a list of `Product` objects that have been bought together (that is, belong to the same order).

In this method, you perform the following tasks:

1. You get the product IDs for the given `Product` objects.
2. You iterate over the product IDs. For each ID, you iterate again over the product IDs and skip the same product so that you get the products that are bought together with each product.
3. You get the Redis product key for each product bought using the `get_product_id()` method. For a product with an ID of `33`, this method returns the key `product:33:purchased_with`. This is the key for the sorted set that contains the product IDs of products that were bought together with this one.
4. You increment the score of each product ID contained in the sorted set by 1. The score represents the number of times another product has been bought together with the given product.

You now have a method to store and score the products that were bought together. Next, you need a method to retrieve the products that were bought together for a list of given products. Add the following `suggest_products_for()` method to the `Recommender` class:

```
def suggest_products_for(self, products, max_results=6):
    product_ids = [p.id for p in products]
    if len(products) == 1:
        # only 1 product
        suggestions = r.zrange(
                     self.get_product_key(product_ids[0]),
                     0, -1, desc=True)[:max_results]
    else:
        # generate a temporary key
        flat_ids = ''.join([str(id) for id in product_ids])
        tmp_key = f'tmp_{flat_ids}'
```

```
            # multiple products, combine scores of all products
            # store the resulting sorted set in a temporary key
            keys = [self.get_product_key(id) for id in product_ids]
            r.zunionstore(tmp_key, keys)
            # remove ids for the products the recommendation is for
            r.zrem(tmp_key, *product_ids)
            # get the product ids by their score, descendant sort
            suggestions = r.zrange(tmp_key, 0, -1,
                                   desc=True)[:max_results]
            # remove the temporary key
            r.delete(tmp_key)
        suggested_products_ids = [int(id) for id in suggestions]
        # get suggested products and sort by order of appearance
        suggested_products = list(Product.objects.filter(
            id__in=suggested_products_ids))
        suggested_products.sort(key=lambda x: suggested_products_ids.index(x.id))
        return suggested_products
```

The `suggest_products_for()` method receives the following parameters:

- `products`: This is a list of `Product` objects to get recommendations for. It can contain one or more products.
- `max_results`: This is an integer that represents the maximum number of recommendations to return.

In this method, you perform the following actions:

1. You get the product IDs for the given `Product` objects.
2. If only one product is given, you retrieve the ID of the products that were bought together with the given product, ordered by the total number of times that they were bought together. To do so, you use Redis' `ZRANGE` command. You limit the number of results to the number specified in the `max_results` attribute (`6` by default).
3. If more than one product is given, you generate a temporary Redis key built with the IDs of the products.
4. Combine and sum all scores for the items contained in the sorted set of each of the given products. This is done using the Redis `ZUNIONSTORE` command. The `ZUNIONSTORE` command performs a union of the sorted sets with the given keys and stores the aggregated sum of scores of the elements in a new Redis key. You can read more about this command at `https://redis.io/commands/zunionstore/`. You save the aggregated scores in the temporary key.
5. Since you are aggregating scores, you might obtain the same products you are getting recommendations for. You remove them from the generated sorted set using the `ZREM` command.

6. You retrieve the IDs of the products from the temporary key, ordered by their scores using the `ZRANGE` command. You limit the number of results to the number specified in the `max_results` attribute. Then, you remove the temporary key.
7. Finally, you get the `Product` objects with the given IDs, and you order the products in the same order as them.

For practical purposes, let's also add a method to clear the recommendations. Add the following method to the `Recommender` class:

```
def clear_purchases(self):
    for id in Product.objects.values_list('id', flat=True):
        r.delete(self.get_product_key(id))
```

Let's try the recommendation engine. Make sure you include several `Product` objects in the database and initialize the Redis Docker container using the following command:

```
docker run -it --rm --name redis -p 6379:6379 redis
```

Open another shell and run the following command to open the Python shell:

```
python manage.py shell
```

Make sure that you have at least four different products in your database. Retrieve four different products by their names:

```
>>> from shop.models import Product
>>> black_tea = Product.objects.get(name='Black tea')
>>> red_tea = Product.objects.get(name='Red tea')
>>> green_tea = Product.objects.get(name='Green tea')
>>> tea_powder = Product.objects.get(name='Tea powder')
```

Then, add some test purchases to the recommendation engine:

```
>>> from shop.recommender import Recommender
>>> r = Recommender()
>>> r.products_bought([black_tea, red_tea])
>>> r.products_bought([black_tea, green_tea])
>>> r.products_bought([red_tea, black_tea, tea_powder])
>>> r.products_bought([green_tea, tea_powder])
>>> r.products_bought([black_tea, tea_powder])
>>> r.products_bought([red_tea, green_tea])
```

You have stored the following scores:

```
black_tea:  red_tea (2), tea_powder (2), green_tea (1)
red_tea:    black_tea (2), tea_powder (1), green_tea (1)
green_tea:  black_tea (1), tea_powder (1), red_tea(1)
tea_powder: black_tea (2), red_tea (1), green_tea (1)
```

This is a representation of products that have been bought together with each of the products, including how many times they have been bought together.

Let's retrieve product recommendations for a single product:

```
>>> r.suggest_products_for([black_tea])
[<Product: Tea powder>, <Product: Red tea>, <Product: Green tea>]
>>> r.suggest_products_for([red_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Green tea>]
>>> r.suggest_products_for([green_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Red tea>]
>>> r.suggest_products_for([tea_powder])
[<Product: Black tea>, <Product: Red tea>, <Product: Green tea>]
```

You can see that the order for recommended products is based on their score. Let's get recommendations for multiple products with aggregated scores:

```
>>> r.suggest_products_for([black_tea, red_tea])
[<Product: Tea powder>, <Product: Green tea>]
>>> r.suggest_products_for([green_tea, red_tea])
[<Product: Black tea>, <Product: Tea powder>]
>>> r.suggest_products_for([tea_powder, black_tea])
[<Product: Red tea>, <Product: Green tea>]
```

You can see that the order of the suggested products matches the aggregated scores. For example, products suggested for `black_tea` and `red_tea` are `tea_powder` (2+1) and `green_tea` (1+1).

You have verified that your recommendation algorithm works as expected. Let's now display recommendations for products on your site.

Edit the `views.py` file of the `shop` application. Add the functionality to retrieve a maximum of four recommended products into the `product_detail` view, as follows:

```
from .recommender import Recommender

def product_detail(request, id, slug):
    product = get_object_or_404(Product,
                                id=id,
                                slug=slug,
                                available=True)
    cart_product_form = CartAddProductForm()
    r = Recommender()
    recommended_products = r.suggest_products_for([product], 4)
    return render(request,
                  'shop/product/detail.html',
                  {'product': product,
                   'cart_product_form': cart_product_form,
                   'recommended_products': recommended_products})
```

Edit the `shop/product/detail.html` template of the `shop` application and add the following code after `{{ product.description|linebreaks }}`:

```
{% if recommended_products %}
  <div class="recommendations">
    <h3>People who bought this also bought</h3>
    {% for p in recommended_products %}
      <div class="item">
        <a href="{{ p.get_absolute_url }}">
          <img src="{% if p.image %}{{ p.image.url }}{% else %}
          {% static  "img/no_image.png" %}{% endif %}">
        </a>
        <p><a href="{{ p.get_absolute_url }}">{{ p.name }}</a></p>
      </div>
    {% endfor %}
  </div>
{% endif %}
```

Run the development server, and open `http://127.0.0.1:8000/` in your browser. Click on any product to view its details. You should see that recommended products are displayed below the product, as shown in *Figure 10.14*:

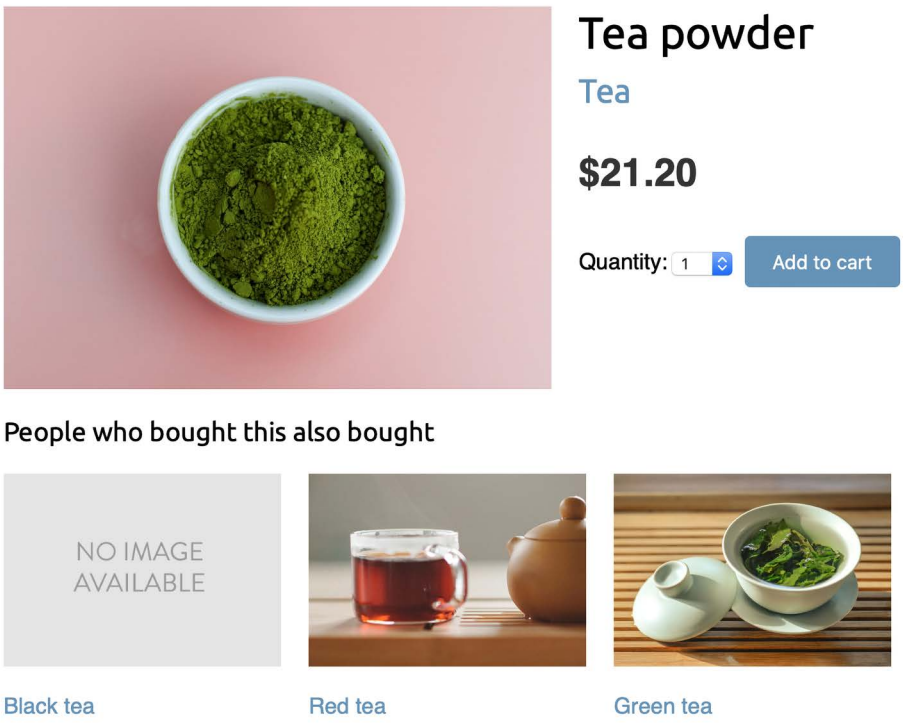


Figure 10.14: The product detail page, including recommended products

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

You are also going to include product recommendations in the cart. The recommendations will be based on the products that the user has added to the cart.

Edit `views.py` inside the `cart` application, import the `Recommender` class, and edit the `cart_detail` view to make it look like the following:

```
from shop.recommender import Recommender

def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(initial={
                            'quantity': item['quantity'],
                            'override': True})
    coupon_apply_form = CouponApplyForm()

    r = Recommender()
    cart_products = [item['product'] for item in cart]
    if(cart_products):
        recommended_products = r.suggest_products_for(
                                    cart_products,
                                    max_results=4)
    else:
        recommended_products = []
    return render(request,
                  'cart/detail.html',
                  {'cart': cart,
                   'coupon_apply_form': coupon_apply_form,
                   'recommended_products': recommended_products})
```

Edit the `cart/detail.html` template of the `cart` application and add the following code just after the `</table>` HTML tag:

```
{% if recommended_products %}
  <div class="recommendations cart">
    <h3>People who bought this also bought</h3>
    {% for p in recommended_products %}
```

```
      <div class="item">
        <a href="{{ p.get_absolute_url }}">
          <img src="{% if p.image %}{{ p.image.url }}{% else %}
          {% static "img/no_image.png" %}{% endif %}">
        </a>
        <p><a href="{{ p.get_absolute_url }}">{{ p.name }}</a></p>
      </div>
    {% endfor %}
  </div>
{% endif %}
```

Open `http://127.0.0.1:8000/en/` in your browser and add a couple of products to your cart. When you navigate to `http://127.0.0.1:8000/en/cart/`, you should see the aggregated product recommendations for the items in the cart, as follows:

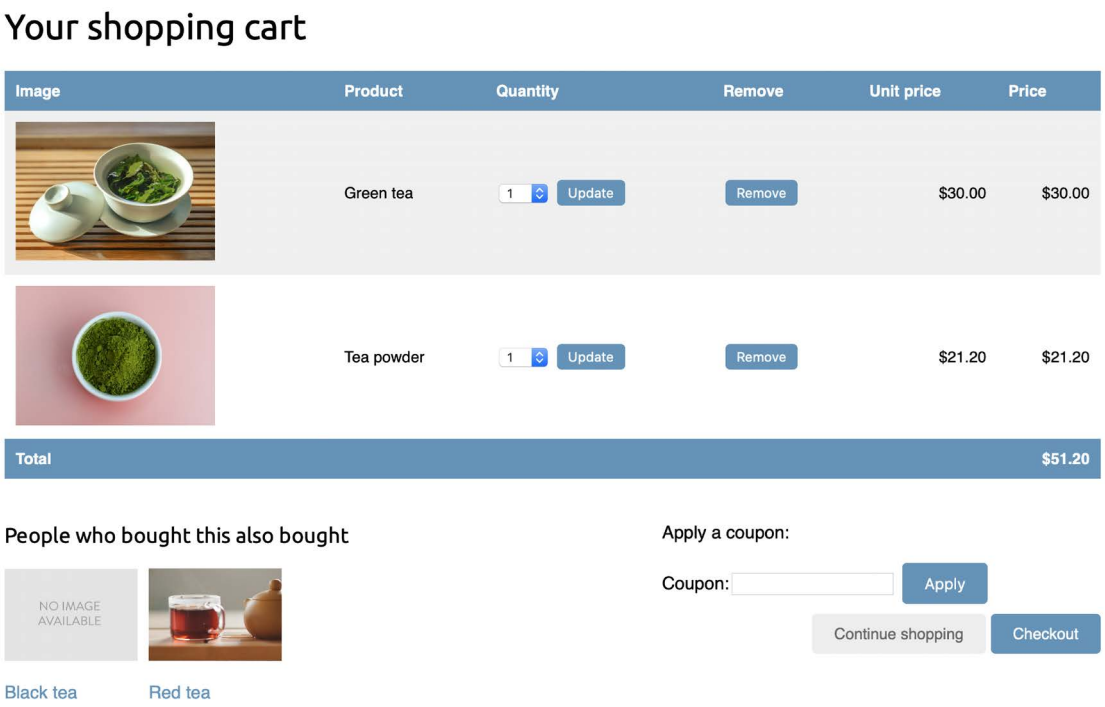


Figure 10.15: The shopping cart details page, including recommended products

Congratulations! You have built a complete recommendation engine using Django and Redis.

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-4-by-example/tree/main/Chapter10`
- Discounts for Stripe Checkout – `https://stripe.com/docs/payments/checkout/discounts`
- The Redis `ZUNIONSTORE` command – `https://redis.io/commands/zunionstore/`

Summary

In this chapter, you created a coupon system using Django sessions and integrated it with Stripe. You also built a recommendation engine using Redis to recommend products that are usually purchased together.

The next chapter will give you an insight into the internationalization and localization of Django projects. You will learn how to translate code and manage translations with Rosetta. You will implement URLs for translations and build a language selector. You will also implement model translations using `django-parler` and you will validate localized form fields using `django-localflavor`.

11

Adding Internationalization to Your Shop

In the previous chapter, you added a coupon system to your shop and built a product recommendation engine.

In this chapter, you will learn how internationalization and localization work.

This chapter will cover the following points:

- Preparing your project for internationalization
- Managing translation files
- Translating Python code
- Translating templates
- Using Rosetta to manage translations
- Translating URL patterns and using a language prefix in URLs
- Allowing users to switch language
- Translating models using `django-parler`
- Using translations with the ORM
- Adapting views to use translations
- Using localized form fields of `django-localflavor`

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter11`.

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

Internationalization with Django

Django offers full internationalization and localization support. It allows you to translate your application into multiple languages and it handles locale-specific formatting for dates, times, numbers, and time zones. Let's clarify the difference between internationalization and localization:

Internationalization (frequently abbreviated to **i18n**) is the process of adapting software for the potential use of different languages and locales so that it isn't hardwired to a specific language or locale.

Localization (abbreviated to **l10n**) is the process of actually translating the software and adapting it to a particular locale. Django itself is translated into more than 50 languages using its internationalization framework.

The internationalization framework allows you to easily mark strings for translation, both in Python code and in your templates. It relies on the GNU gettext toolset to generate and manage message files. A **message file** is a plain text file that represents a language. It contains a part, or all, of the translation strings found in your application and their respective translations for a single language. Message files have the `.po` extension. Once the translation is done, message files are compiled to offer rapid access to translated strings. The compiled translation files have the `.mo` extension.

Internationalization and localization settings

Django provides several settings for internationalization. The following settings are the most relevant ones:

- `USE_I18N`: A Boolean that specifies whether Django's translation system is enabled. This is `True` by default.
- `USE_L10N`: A Boolean indicating whether localized formatting is enabled. When active, localized formats are used to represent dates and numbers. This is `False` by default.
- `USE_TZ`: A Boolean that specifies whether datetimes are time-zone-aware. When you create a project with the `startproject` command, this is set to `True`.
- `LANGUAGE_CODE`: The default language code for the project. This is in the standard language ID format, for example, `'en-us'` for American English, or `'en-gb'` for British English. This setting requires `USE_I18N` to be set to `True` in order to take effect. You can find a list of valid language IDs at `http://www.i18nguy.com/unicode/language-identifiers.html`.
- `LANGUAGES`: A tuple that contains available languages for the project. They come in two tuples of a **language code** and a **language name**. You can see the list of available languages at `django.conf.global_settings`. When you choose which languages your site will be available in, you set `LANGUAGES` to a subset of that list.
- `LOCALE_PATHS`: A list of directories where Django looks for message files containing translations for the project.
- `TIME_ZONE`: A string that represents the time zone for the project. This is set to `'UTC'` when you create a new project using the `startproject` command. You can set it to any other time zone, such as `'Europe/Madrid'`.

These are some of the internationalization and localization settings available. You can find the full list at https://docs.djangoproject.com/en/4.1/ref/settings/#globalization-i18n-l10n.

Internationalization management commands

Django includes the following management commands to manage translations:

- makemessages: This runs over the source tree to find all the strings marked for translation and creates or updates the .po message files in the locale directory. A single .po file is created for each language.
- compilemessages: This compiles the existing .po message files to .mo files, which are used to retrieve translations.

Installing the gettext toolkit

You will need the gettext toolkit to be able to create, update, and compile message files. Most Linux distributions include the gettext toolkit. If you are using macOS, the simplest way to install it is via Homebrew, at https://brew.sh/, with the following command:

```
brew install gettext
```

You might also need to force link it with the following command:

```
brew link --force gettext
```

If you are using Windows, follow the steps at https://docs.djangoproject.com/en/4.1/topics/i18n/translation/#gettext-on-windows. You can download a precompiled gettext binary installer for Windows from https://mlocati.github.io/articles/gettext-iconv-windows.html.

How to add translations to a Django project

Let's take a look at the process of internationalizing your project. You will need to do the following:

1. Mark the strings for translation in your Python code and your templates.
2. Run the makemessages command to create or update message files that include all the translation strings from your code.
3. Translate the strings contained in the message files and compile them using the compilemessages management command.

How Django determines the current language

Django comes with a middleware that determines the current language based on the request data. This is the LocaleMiddleware middleware that resides in django.middleware.locale.LocaleMiddleware, which performs the following tasks:

1. If you are using i18n_patterns, that is, you are using translated URL patterns, it looks for a language prefix in the requested URL to determine the current language.
2. If no language prefix is found, it looks for an existing LANGUAGE_SESSION_KEY in the current user's session.

3. If the language is not set in the session, it looks for an existing cookie with the current language. A custom name for this cookie can be provided in the `LANGUAGE_COOKIE_NAME` setting. By default, the name for this cookie is `django_language`.
4. If no cookie is found, it looks for the `Accept-Language` HTTP header of the request.
5. If the `Accept-Language` header does not specify a language, Django uses the language defined in the `LANGUAGE_CODE` setting.

By default, Django will use the language defined in the `LANGUAGE_CODE` setting unless you are using `LocaleMiddleware`. The process described here only applies when using this middleware.

Preparing your project for internationalization

Let's prepare your project to use different languages. You are going to create an English and a Spanish version for your shop. Edit the `settings.py` file of your project and add the following `LANGUAGES` setting to it. Place it next to the `LANGUAGE_CODE` setting:

```
LANGUAGES = [
    ('en', 'English'),
    ('es', 'Spanish'),
]
```

The `LANGUAGES` setting contains two tuples that consist of a language code and a name. Language codes can be locale-specific, such as `en-us` or `en-gb`, or generic, such as `en`. With this setting, you specify that your application will only be available in English and Spanish. If you don't define a custom `LANGUAGES` setting, the site will be available in all the languages that Django is translated into.

Make your `LANGUAGE_CODE` setting look like the following:

```
LANGUAGE_CODE = 'en'
```

Add `'django.middleware.locale.LocaleMiddleware'` to the `MIDDLEWARE` setting. Make sure that this middleware comes after `SessionMiddleware` because `LocaleMiddleware` needs to use session data. It also has to be placed before `CommonMiddleware` because the latter needs an active language to resolve the requested URL. The `MIDDLEWARE` setting should now look like the following:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

The order of middleware classes is very important because each middleware can depend on data set by another middleware that was executed previously. Middleware is applied for requests in order of appearance in `MIDDLEWARE`, and in reverse order for responses.

Create the following directory structure inside the main project directory, next to the `manage.py` file:

```
locale/
    en/
    es/
```

The `locale` directory is the place where message files for your application will reside. Edit the `settings.py` file again and add the following setting to it:

```
LOCALE_PATHS = [
    BASE_DIR / 'locale',
]
```

The `LOCALE_PATHS` setting specifies the directories where Django has to look for translation files. Locale paths that appear first have the highest precedence.

When you use the `makemessages` command from your project directory, message files will be generated in the `locale/` path you created. However, for applications that contain a `locale/` directory, message files will be generated in that directory.

Translating Python code

To translate literals in your Python code, you can mark strings for translation using the `gettext()` function included in `django.utils.translation`. This function translates the message and returns a string. The convention is to import this function as a shorter alias named _ (the underscore character).

You can find all the documentation about translations at `https://docs.djangoproject.com/en/4.1/topics/i18n/translation/`.

Standard translations

The following code shows how to mark a string for translation:

```
from django.utils.translation import gettext as _
output = _('Text to be translated.')
```

Lazy translations

Django includes **lazy** versions for all of its translation functions, which have the suffix `_lazy()`. When using the lazy functions, strings are translated when the value is accessed, rather than when the function is called (this is why they are translated **lazily**). The lazy translation functions come in handy when the strings marked for translation are in paths that are executed when modules are loaded.

Using `gettext_lazy()` instead of `gettext()` means that strings are translated when the value is accessed. Django offers a lazy version for all translation functions.

Translations including variables

The strings marked for translation can include placeholders to include variables in the translations. The following code is an example of a translation string with a placeholder:

```
from django.utils.translation import gettext as _
month = _('April')
day = '14'
output = _('Today is %(month)s %(day)s') % {'month': month,
                                            'day': day}
```

By using placeholders, you can reorder the text variables. For example, an English translation of the previous example might be *today is April 14*, while the Spanish one might be *hoy es 14 de Abril*. Always use string interpolation instead of positional interpolation when you have more than one parameter for the translation string. By doing so, you will be able to reorder the placeholder text.

Plural forms in translations

For plural forms, you can use `ngettext()` and `ngettext_lazy()`. These functions translate singular and plural forms depending on an argument that indicates the number of objects. The following example shows how to use them:

```
output = ngettext('there is %(count)d product',
                  'there are %(count)d products',
                  count) % {'count': count}
```

Now that you know the basics of translating literals in your Python code, it's time to apply translations to your project.

Translating your own code

Edit the `settings.py` file of your project, import the `gettext_lazy()` function, and change the `LANGUAGES` setting, as follows, to translate the language names:

```
from django.utils.translation import gettext_lazy as _
# ...

LANGUAGES = [
    ('en', _('English')),
    ('es', _('Spanish')),
]
```

Here, you use the `gettext_lazy()` function instead of `gettext()` to avoid a circular import, thus translating the languages' names when they are accessed.

Open the shell and run the following command from your project directory:

```
django-admin makemessages --all
```

You should see the following output:

```
processing locale es
processing locale en
```

Take a look at the `locale/` directory. You should see a file structure like the following:

```
en/
    LC_MESSAGES/
        django.po
es/
    LC_MESSAGES/
        django.po
```

A `.po` message file has been created for each language. Open `es/LC_MESSAGES/django.po` with a text editor. At the end of the file, you should be able to see the following:

```
#: myshop/settings.py:118
msgid "English"
msgstr ""
#: myshop/settings.py:119
msgid "Spanish"
msgstr ""
```

Each translation string is preceded by a comment showing details about the file and the line where it was found. Each translation includes two strings:

- `msgid`: The translation string as it appears in the source code.
- `msgstr`: The language translation, which is empty by default. This is where you have to enter the actual translation for the given string.

Fill in the `msgstr` translations for the given `msgid` string, as follows:

```
#: myshop/settings.py:118
msgid "English"
msgstr "Inglés"
#: myshop/settings.py:119
msgid "Spanish"
msgstr "Español"
```

Save the modified message file, open the shell, and run the following command:

```
django-admin compilemessages
```

If everything goes well, you should see an output like the following:

```
processing file django.po in myshop/locale/en/LC_MESSAGES
processing file django.po in myshop/locale/es/LC_MESSAGES
```

The output gives you information about the message files that are being compiled. Take a look at the `locale` directory of the `myshop` project again. You should see the following files:

```
en/
    LC_MESSAGES/
        django.mo
        django.po
es/
    LC_MESSAGES/
        django.mo
        django.po
```

You can see that a `.mo` compiled message file has been generated for each language.

You have translated the language names. Now, let's translate the model field names that are displayed on the site. Edit the `models.py` file of the `orders` application, and add names marked for translation to the `Order` model fields as follows:

```
from django.utils.translation import gettext_lazy as _

class Order(models.Model):
    first_name = models.CharField(_('first name'),
                                  max_length=50)
    last_name = models.CharField(_('last name'),
                                 max_length=50)
    email = models.EmailField(_('e-mail'))
    address = models.CharField(_('address'),
                               max_length=250)
    postal_code = models.CharField(_('postal code'),
                                   max_length=20)
    city = models.CharField(_('city'),
                            max_length=100)
    # ...
```

You have added names for the fields that are displayed when a user is placing a new order. These are `first_name`, `last_name`, `email`, `address`, `postal_code`, and `city`. Remember that you can also use the `verbose_name` attribute to name the fields.

Create the following directory structure inside the `orders` application directory:

```
locale/
    en/
    es/
```

By creating a `locale` directory, the translation strings of this application will be stored in a message file under this directory instead of the main messages file. In this way, you can generate separate translation files for each application.

Open the shell from the project directory and run the following command:

```
django-admin makemessages --all
```

You should see the following output:

```
processing locale es
processing locale en
```

Open the `locale/es/LC_MESSAGES/django.po` file of the `order` application using a text editor. You will see the translation strings for the `Order` model. Fill in the following `msgstr` translations for the given `msgid` strings:

```
#: orders/models.py:12
msgid "first name"
msgstr "nombre"
#: orders/models.py:14
msgid "last name"
msgstr "apellidos"
#: orders/models.py:16
msgid "e-mail"
msgstr "e-mail"
#: orders/models.py:17
msgid "address"
msgstr "dirección"
#: orders/models.py:19
msgid "postal code"
msgstr "código postal"
#: orders/models.py:21
msgid "city"
msgstr "ciudad"
```

After you have finished adding the translations, save the file.

Besides a text editor, you can use Poedit to edit translations. Poedit is a piece of software for editing translations that uses gettext. It is available for Linux, Windows, and macOS. You can download Poedit from `https://poedit.net/`.

Let's also translate the forms of your project. The `OrderCreateForm` of the `orders` application does not have to be translated. That's because it is a `ModelForm` and it uses the `verbose_name` attribute of the `Order` model fields for the form field labels. You are going to translate the forms of the `cart` and `coupons` applications.

Edit the `forms.py` file inside the `cart` application directory and add a `label` attribute to the `quantity` field of the `CartAddProductForm`. Then, mark this field for translation, as follows:

```
from django import forms
from django.utils.translation import gettext_lazy as _

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,
                    label=_('Quantity'))
    override = forms.BooleanField(required=False,
                                  initial=False,
                                  widget=forms.HiddenInput)
```

Edit the `forms.py` file of the `coupons` application and translate the `CouponApplyForm` form, as follows:

```
from django import forms
from django.utils.translation import gettext_lazy as _

class CouponApplyForm(forms.Form):
    code = forms.CharField(label=_('Coupon'))
```

You have added a label to the `code` field and marked it for translation.

Translating templates

Django offers the `{% trans %}` and `{% blocktrans %}` template tags to translate the strings in templates. In order to use the translation template tags, you have to add `{% load i18n %}` to the top of your template to load them.

The {% trans %} template tag

The `{% trans %}` template tag allows you to mark a literal for translation. Internally, Django executes `gettext()` on the given text. This is how to mark a string for translation in a template:

```
{% trans "Text to be translated" %}
```

You can use `as` to store the translated content in a variable that you can use throughout your template. The following example stores the translated text in a variable called `greeting`:

```
{% trans "Hello!" as greeting %}
<h1>{{ greeting }}</h1>
```

The `{% trans %}` tag is useful for simple translation strings, but it can't handle content for translation that includes variables.

The {% blocktrans %} template tag

The `{% blocktrans %}` template tag allows you to mark content that includes literals and variable content using placeholders. The following example shows you how to use the `{% blocktrans %}` tag, including a `name` variable in the content for translation:

```
{% blocktrans %}Hello {{ name }}!{% endblocktrans %}
```

You can use `with` to include template expressions, such as accessing object attributes or applying template filters to variables. You always have to use placeholders for these. You can't access expressions or object attributes inside the `blocktrans` block. The following example shows you how to use `with` to include an object attribute to which the `capfirst` filter has been applied:

```
{% blocktrans with name=user.name|capfirst %}
  Hello {{ name }}!
{% endblocktrans %}
```

Use the `{% blocktrans %}` tag instead of `{% trans %}` when you need to include variable content in your translation string.

Translating the shop templates

Edit the `shop/base.html` template of the `shop` application. Make sure that you load the `i18n` tag at the top of the template and mark the strings for translation, as follows. New code is highlighted in bold:

```
{% load i18n %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>
    {% block title %}{% trans "My shop" %}{% endblock %}
  </title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
```

```
</head>
<body>
  <div id="header">
    <a href="/" class="logo">{% trans "My shop" %}</a>
  </div>
  <div id="subheader">
    <div class="cart">
      {% with total_items=cart|length %}
        {% if total_items > 0 %}
          {% trans "Your cart" %}:
          <a href="{% url "cart:cart_detail" %}">
            {% blocktrans with total=cart.get_total_price count items=total_
items %}
              {{ items }} item, ${{ total }}
            {% plural %}
              {{ items }} items, ${{ total }}
            {% endblocktrans %}
          </a>
        {% elif not order %}
          {% trans "Your cart is empty." %}
        {% endif %}
      {% endwith %}
    </div>
  </div>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
</body>
</html>
```

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

Notice the `{% blocktrans %}` tag to display the cart's summary. The cart's summary was previously as follows:

```
{{ total_items }} item{{ total_items|pluralize }},
${{ cart.get_total_price }}
```

You changed it, and now you use `{% blocktrans with ... %}` to set up the placeholder `total` with the value of `cart.get_total_price` (the object method called here). You also use `count`, which allows you to set a variable for counting objects for Django to select the right plural form. You set the `items` variable to count objects with the value of `total_items`.

This allows you to set a translation for the singular and plural forms, which you separate with the `{% plural %}` tag within the `{% blocktrans %}` block. The resulting code is:

```
{% blocktrans with total=cart.get_total_price count items=total_items %}
  {{ items }} item, ${{ total }}
{% plural %}
  {{ items }} items, ${{ total }}
{% endblocktrans %}
```

Next, edit the `shop/product/detail.html` template of the `shop` application and load the `i18n` tags at the top of it, but after the `{% extends %}` tag, which always has to be the first tag in the template:

```
{% extends "shop/base.html" %}
{% load i18n %}
{% load static %}
...
```

Then, find the following line:

```
<input type="submit" value="Add to cart">
```

Replace it with the following:

```
<input type="submit" value="{% trans "Add to cart" %}">
```

Then, find the following line:

```
<h3>People who bought this also bought</h3>
```

Replace it with the following:

```
<h3>{% trans "People who bought this also bought" %}</h3>
```

Now, translate the `orders` application template. Edit the `orders/order/create.html` template of the `orders` application and mark the text for translation, as follows:

```
{% extends "shop/base.html" %}
{% load i18n %}
{% block title %}
  {% trans "Checkout" %}
{% endblock %}
{% block content %}
  <h1>{% trans "Checkout" %}</h1>
  <div class="order-info">
    <h3>{% trans "Your order" %}</h3>
    <ul>
      {% for item in cart %}
        <li>
          {{ item.quantity }}x {{ item.product.name }}
```

```
            <span>${{ item.total_price }}</span>
          </li>
        {% endfor %}
        {% if cart.coupon %}
          <li>
            {% blocktrans with code=cart.coupon.code discount=cart.coupon.
discount %}
              "{{ code }}" ({{ discount }}% off)
            {% endblocktrans %}
            <span class="neg">- ${{ cart.get_discount|floatformat:2 }}</span>
          </li>
        {% endif %}
      </ul>
      <p>{% trans "Total" %}: ${{
      cart.get_total_price_after_discount|floatformat:2 }}</p>
    </div>
    <form method="post" class="order-form">
      {{ form.as_p }}
      <p><input type="submit" value="{% trans "Place order" %}"></p>
      {% csrf_token %}
    </form>
{% endblock %}
```

Make sure that no template tag is split across multiple lines. Take a look at the following files in the code that accompanies this chapter to see how the strings have been marked for translation:

- The `shop` application: Template `shop/product/list.html`
- The `orders` application: Template `orders/order/pdf.html`
- The `cart` application: Template `cart/detail.html`
- The `payments` application: Templates `payment/process.html`, `payment/completed.html`, and `payment/canceled.html`

Remember that you can find the source code for this chapter at `https://github.com/PacktPublishing/Django-4-by-Example/tree/master/Chapter11`.

Let's update the message files to include the new translation strings. Open the shell and run the following command:

```
django-admin makemessages --all
```

The `.po` files are inside the `locale` directory of the `myshop` project, and you'll see that the `orders` application now contains all the strings that you marked for translation.

Edit the `.po` translation files of the project and the `orders` application and include Spanish translations in `msgstr`. You can also use the translated `.po` files in the source code that accompanies this chapter.

Run the following command to compile the translation files:

```
django-admin compilemessages
```

You will see the following output:

```
processing file django.po in myshop/locale/en/LC_MESSAGES
processing file django.po in myshop/locale/es/LC_MESSAGES
processing file django.po in myshop/orders/locale/en/LC_MESSAGES
processing file django.po in myshop/orders/locale/es/LC_MESSAGES
```

A `.mo` file containing compiled translations has been generated for each `.po` translation file.

Using the Rosetta translation interface

Rosetta is a third-party application that allows you to edit translations using the same interface as the Django administration site. Rosetta makes it easy to edit `.po` files, and it updates compiled translation files. Let's add it to your project.

Install Rosetta via `pip` using this command:

```
pip install django-rosetta==0.9.8
```

Then, add `'rosetta'` to the `INSTALLED_APPS` setting in your project's `settings.py` file, as follows:

```
INSTALLED_APPS = [
    # ...
    'rosetta',
]
```

You need to add Rosetta's URLs to your main URL configuration. Edit the main `urls.py` file of your project and add the following URL pattern highlighted in bold:

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

Make sure you place it before the `shop.urls` pattern to avoid an undesired pattern match.

Open `http://127.0.0.1:8000/admin/` and log in with a superuser. Then, navigate to `http://127.0.0.1:8000/rosetta/` in your browser. In the **Filter** menu, click **THIRD PARTY** to display all the available message files, including those that belong to the `orders` application.

You should see a list of existing languages, as follows:

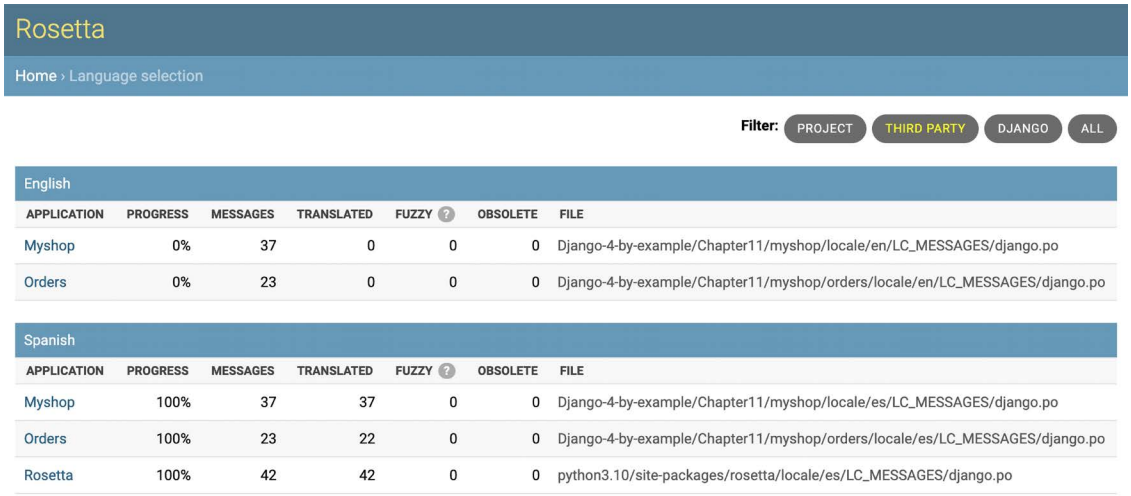


Figure 11.1: The Rosetta administration interface

Click the **Myshop** link under the **Spanish** section to edit the Spanish translations. You should see a list of translation strings, as follows:

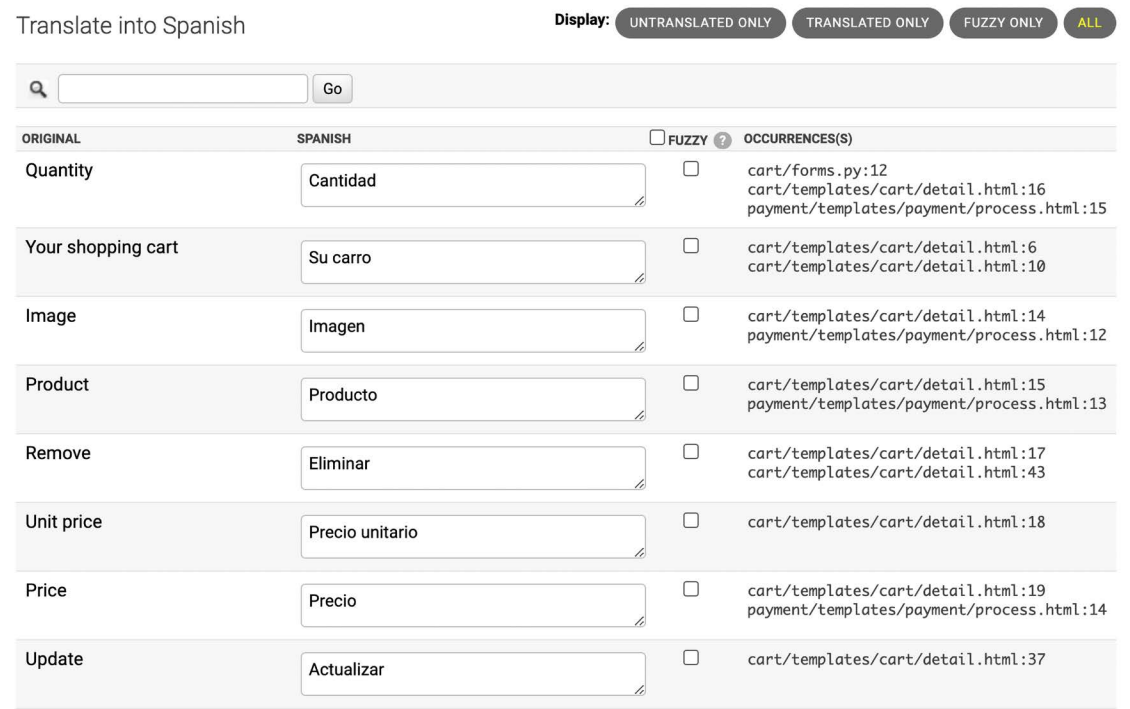


Figure 11.2: Editing Spanish translations using Rosetta

You can enter the translations under the **SPANISH** column. The **OCCURRENCE(S)** column displays the files and lines of code where each translation string was found.

Translations that include placeholders will appear as follows:

Figure 11.3: Translations including placeholders

Rosetta uses a different background color to display placeholders. When you translate content, make sure that you keep placeholders untranslated. For example, take the following string:

```
%(items)s items, $%(total)s
```

It can be translated into Spanish as follows:

```
%(items)s productos, $%(total)s
```

You can take a look at the source code that comes with this chapter to use the same Spanish translations for your project.

When you finish editing translations, click the **Save and translate next block** button to save the translations to the `.po` file. Rosetta compiles the message file when you save translations, so there is no need for you to run the `compilemessages` command. However, Rosetta requires write access to the `locale` directories to write the message files. Make sure that the directories have valid permissions.

If you want other users to be able to edit translations, open `http://127.0.0.1:8000/admin/auth/group/add/` in your browser and create a new group named `translators`. Then, access `http://127.0.0.1:8000/admin/auth/user/` to edit the users to whom you want to grant permissions so that they can edit translations. When editing a user, under the **Permissions** section, add the `translators` group to the **Chosen Groups** for each user. Rosetta is only available to superusers or users who belong to the `translators` group.

You can read Rosetta's documentation at `https://django-rosetta.readthedocs.io/`.

When you add new translations to your production environment, if you serve Django with a real web server, you will have to reload your server after running the `compilemessages` command, or after saving the translations with Rosetta, for any changes to take effect.

When editing translations, a translation can be marked as *fuzzy*. Let's review what fuzzy translations are.

Fuzzy translations

When editing translations in Rosetta, you can see a **FUZZY** column. This is not a Rosetta feature; it is provided by gettext. If the **FUZZY** flag is active for a translation, it will not be included in the compiled message files. This flag marks translation strings that need to be reviewed by a translator. When `.po` files are updated with new translation strings, it is possible that some translation strings will automatically be flagged as fuzzy. This happens when gettext finds some `msgid` that has been slightly modified. gettext pairs it with what it thinks was the old translation and flags it as fuzzy for review. The translator should then review the fuzzy translations, remove the **FUZZY** flag, and compile the translation file again.

URL patterns for internationalization

Django offers internationalization capabilities for URLs. It includes two main features for internationalized URLs:

- **Language prefix in URL patterns**: Adding a language prefix to URLs to serve each language version under a different base URL.
- **Translated URL patterns**: Translating URL patterns so that every URL is different for each language.

One reason for translating URLs is to optimize your site for search engines. By adding a language prefix to your patterns, you will be able to index a URL for each language instead of a single URL for all of them. Furthermore, by translating URLs into each language, you will provide search engines with URLs that will rank better for each language.

Adding a language prefix to URL patterns

Django allows you to add a language prefix to your URL patterns. For example, the English version of your site can be served under a path starting with `/en/`, and the Spanish version under `/es/`. To use languages in URL patterns, you have to use the `LocaleMiddleware` provided by Django. The framework will use it to identify the current language from the requested URL. Previously, you added it to the `MIDDLEWARE` setting of your project, so you don't need to do it now.

Let's add a language prefix to your URL patterns. Edit the main `urls.py` file of the `myshop` project and add `i18n_patterns()`, as follows:

```
from django.conf.urls.i18n import i18n_patterns

urlpatterns = i18n_patterns(
    path('admin/', admin.site.urls),
    path('cart/', include('cart.urls', namespace='cart')),
    path('orders/', include('orders.urls', namespace='orders')),
    path('payment/', include('payment.urls', namespace='payment')),
    path('coupons/', include('coupons.urls', namespace='coupons')),
    path('rosetta/', include('rosetta.urls')),
    path('', include('shop.urls', namespace='shop')),
)
```

You can combine non-translatable standard URL patterns and patterns under `i18n_patterns` so that some patterns include a language prefix and others don't. However, it's better to use translated URLs only to avoid the possibility that a carelessly translated URL matches a non-translated URL pattern.

Run the development server and open `http://127.0.0.1:8000/` in your browser. Django will perform the steps described in the *How Django determines the current language* section to determine the current language, and it will redirect you to the requested URL, including the language prefix. Take a look at the URL in your browser; it should now look like `http://127.0.0.1:8000/en/`. The current language is the one set by the `Accept-Language` header of your browser if it is Spanish or English; otherwise, it is the default `LANGUAGE_CODE` (English) defined in your settings.

Translating URL patterns

Django supports translated strings in URL patterns. You can use a different translation for each language for a single URL pattern. You can mark URL patterns for translation in the same way as you would with literals, using the `gettext_lazy()` function.

Edit the main `urls.py` file of the `myshop` project and add translation strings to the regular expressions of the URL patterns for the `cart`, `orders`, `payment`, and `coupons` applications, as follows:

```
from django.utils.translation import gettext_lazy as _

urlpatterns = i18n_patterns(
    path('admin/', admin.site.urls),
    path(_('cart/'), include('cart.urls', namespace='cart')),
    path(_('orders/'), include('orders.urls', namespace='orders')),
    path(_('payment/'), include('payment.urls', namespace='payment')),
    path(_('coupons/'), include('coupons.urls', namespace='coupons')),
    path('rosetta/', include('rosetta.urls')),
    path('', include('shop.urls', namespace='shop')),
)
```

Edit the `urls.py` file of the `orders` application and mark the `order_create` URL pattern for translation, as follows:

```
from django.utils.translation import gettext_lazy as _

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

Edit the `urls.py` file of the `payment` application and change the code to the following:

```
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    path(_('process/'), views.payment_process, name='process'),
    path(_('done/'), views.payment_done, name='done'),
    path(_('canceled/'), views.payment_canceled, name='canceled'),
    path('webhook/', webhooks.stripe_webhook, name='stripe-webhook'),
]
```

Note that these URL patterns will include a language prefix because they are included under `i18n_patterns()` in the main `urls.py` file of the project. This will make each URL pattern have a different URI for each available language, one starting with `/en/`, another one with `/es/`, and so on. However, we need a single URL for Stripe to notify events, and we need to avoid language prefixes in the `webhook` URL.

Remove the `webhook` URL pattern from the `urls.py` file of the `payment` application. The file should now look like the following:

```
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    path(_('process/'), views.payment_process, name='process'),
    path(_('done/'), views.payment_done, name='done'),
    path(_('canceled/'), views.payment_canceled, name='canceled'),
]
```

Then, add the following `webhook` URL pattern to the main `urls.py` file of the `myshop` project. The new code is highlighted in bold:

```
from django.utils.translation import gettext_lazy as _
from payment import webhooks

urlpatterns = i18n_patterns(
    path('admin/', admin.site.urls),
    path(_('cart/'), include('cart.urls', namespace='cart')),
    path(_('orders/'), include('orders.urls', namespace='orders')),
    path(_('payment/'), include('payment.urls', namespace='payment')),
    path(_('coupons/'), include('coupons.urls', namespace='coupons')),
    path('rosetta/', include('rosetta.urls')),
    path('', include('shop.urls', namespace='shop')),
)

urlpatterns += [
```

```
    path('payment/webhook/', webhooks.stripe_webhook,
                             name='stripe-webhook'),
]

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

We have added the `webhook` URL pattern to `urlpatterns` outside of `i18n_patterns()` to ensure we maintain a single URL for Stripe event notifications.

You don't need to translate the URL patterns of the `shop` application, as they are built with variables and do not include any other literals.

Open the shell and run the next command to update the message files with the new translations:

```
django-admin makemessages --all
```

Make sure the development server is running with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/en/rosetta/` in your browser and click the **Myshop** link under the **Spanish** section. Click on **UNTRANSLATED ONLY** to only see the strings that have not been translated yet. Now you will see the URL patterns for translation, as shown in *Figure 11.4*:

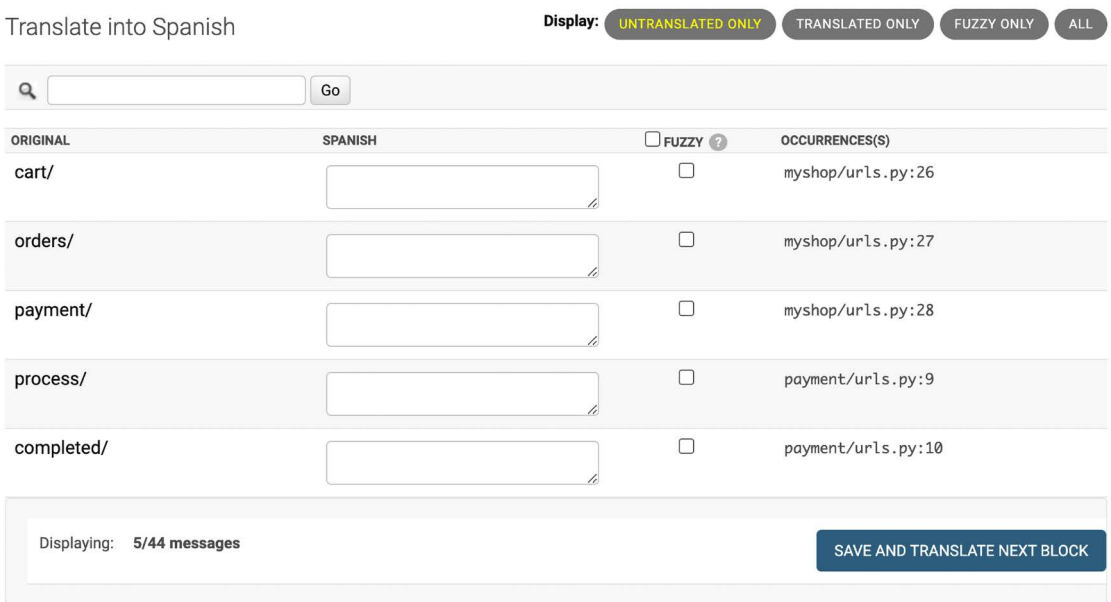


Figure 11.4: URL patterns for translation in the Rosetta interface

Add a different translation string for each URL. Don't forget to include a slash character / at the end of each URL, as shown in *Figure 11.5*:

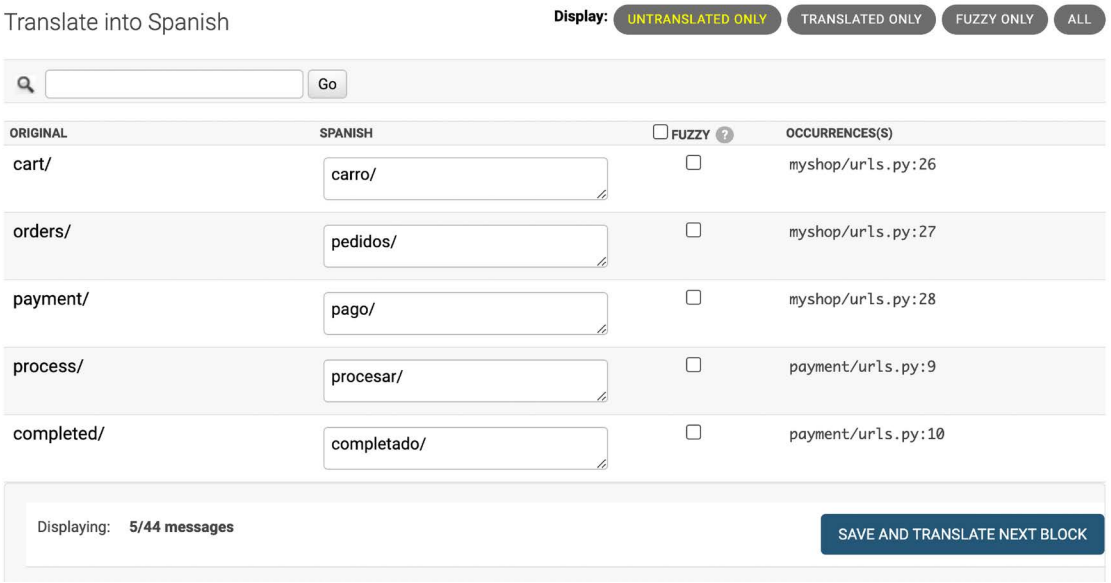


Figure 11.5: Spanish translations for URL patterns in the Rosetta interface

When you have finished, click **SAVE AND TRANSLATE NEXT BLOCK**.

Then, click on **FUZZY ONLY**. You will see translations that have been flagged as fuzzy because they were paired with the old translation of a similar original string. In the case displayed in *Figure 11.6*, the translations are incorrect and need to be corrected:

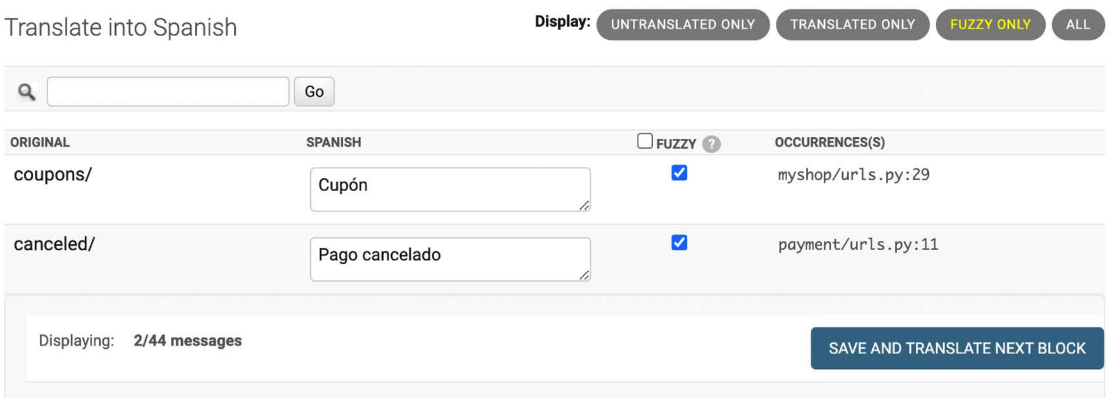


Figure 11.6: Fuzzy translations in the Rosetta interface

Enter the correct text for the fuzzy translations. Rosetta will automatically uncheck the **FUZZY** select box when you enter new text for a translation. When you have finished, click **SAVE AND TRANSLATE NEXT BLOCK**:

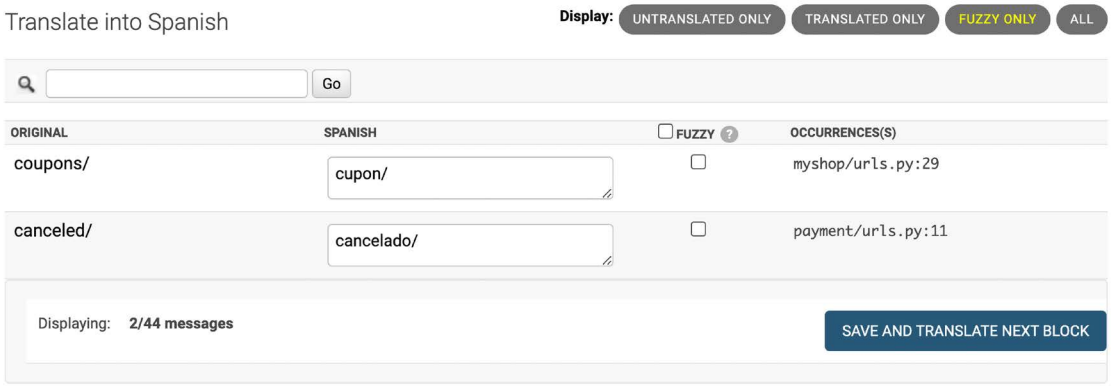


Figure 11.7: Correcting fuzzy translations in the Rosetta interface

You can now go back to `http://127.0.0.1:8000/en/rosetta/files/third-party/` and edit the Spanish translation for the `orders` application as well.

Allowing users to switch language

Since you are serving content that is available in multiple languages, you should let your users switch the site's language. You are going to add a language selector to your site. The language selector will consist of a list of available languages displayed using links.

Edit the `shop/base.html` template of the `shop` application and locate the following lines:

```
<div id="header">
  <a href="/" class="logo">{% trans "My shop" %}</a>
</div>
```

Replace them with the following code:

```
<div id="header">
  <a href="/" class="logo">{% trans "My shop" %}</a>
  {% get_current_language as LANGUAGE_CODE %}
  {% get_available_languages as LANGUAGES %}
  {% get_language_info_list for LANGUAGES as languages %}
  <div class="languages">
    <p>{% trans "Language" %}:</p>
    <ul class="languages">
      {% for language in languages %}
        <li>
          <a href="/{{ language.code }}/"
          {% if language.code == LANGUAGE_CODE %} class="selected"{% endif %}>
            {{ language.name_local }}
          </a>
        </li>
```

```
      {% endfor %}
    </ul>
  </div>
</div>
```

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

This is how you build your language selector:

1. You load the internationalization tags using `{% load i18n %}`.
2. You use the `{% get_current_language %}` tag to retrieve the current language.
3. You get the languages defined in the `LANGUAGES` setting using the `{% get_available_languages %}` template tag.
4. You use the tag `{% get_language_info_list %}` to provide easy access to the language attributes.
5. You build an HTML list to display all available languages, and you add a `selected` class attribute to the current active language.

In the code for the language selector, you used the template tags provided by `i18n`, based on the languages available in the settings of your project. Now open `http://127.0.0.1:8000/` in your browser and take a look. You should see the language selector in the top right-hand corner of the site, as follows:

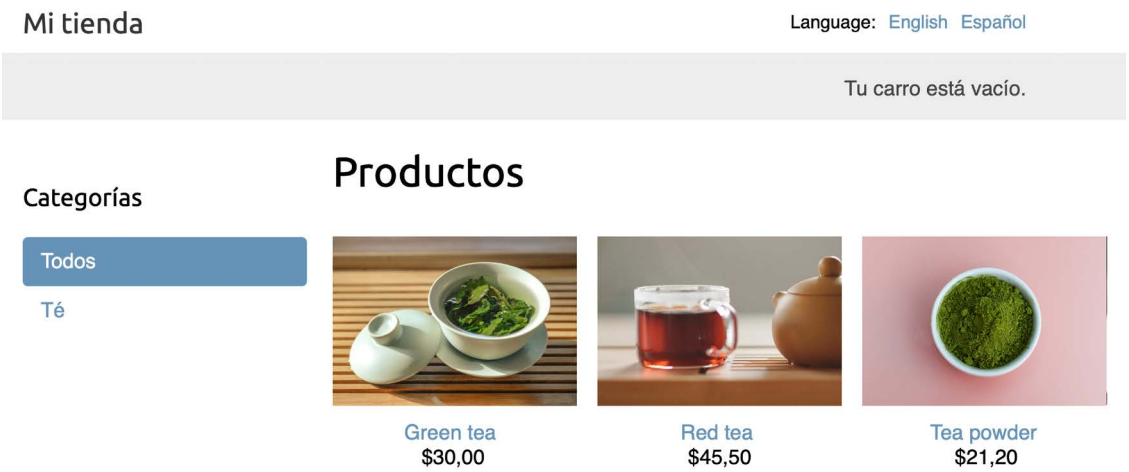


Figure 11.8: The product list page, including a language selector in the site header

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

Users can now easily switch to their preferred language by clicking on it.

Translating models with django-parler

Django does not provide a solution for translating models out of the box. You have to implement your own solution to manage content stored in different languages, or use a third-party module for model translation. There are several third-party applications that allow you to translate model fields. Each of them takes a different approach to storing and accessing translations. One of these applications is `django-parler`. This module offers a very effective way to translate models, and it integrates smoothly with Django's administration site.

`django-parler` generates a separate database table for each model that contains translations. This table includes all the translated fields and a foreign key for the original object that the translation belongs to. It also contains a language field, since each row stores the content for a single language.

Installing django-parler

Install `django-parler` via `pip` using the following command:

```
pip install django-parler==2.3
```

Edit the `settings.py` file of your project and add `'parler'` to the `INSTALLED_APPS` setting, as follows:

```
INSTALLED_APPS = [
    # ...
    'parler',
]
```

Also, add the following code to your settings:

```
# django-parler settings
PARLER_LANGUAGES = {
    None: (
        {'code': 'en'},
        {'code': 'es'},
    ),
    'default': {
        'fallback': 'en',
        'hide_untranslated': False,
    }
}
```

This setting defines the available languages, `en` and `es`, for `django-parler`. You specify the default language `en` and indicate that `django-parler` should not hide untranslated content.

Translating model fields

Let's add translations to your product catalog. `django-parler` provides a `TranslatableModel` model class and a `TranslatedFields` wrapper to translate model fields.

Edit the `models.py` file inside the `shop` application directory and add the following import:

```
from parler.models import TranslatableModel, TranslatedFields
```

Then, modify the `Category` model to make the `name` and `slug` fields translatable, as follows:

```
class Category(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=200),
        slug = models.SlugField(max_length=200,
                                unique=True),
    )
```

The `Category` model now inherits from `TranslatableModel` instead of `models.Model`, and both the `name` and `slug` fields are included in the `TranslatedFields` wrapper.

Edit the `Product` model to add translations for the `name`, `slug`, and `description` fields, as follows:

```
class Product(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=200),
        slug = models.SlugField(max_length=200),
        description = models.TextField(blank=True)
    )
    category = models.ForeignKey(Category,
                                 related_name='products',
                                 on_delete=models.CASCADE)
    image = models.ImageField(upload_to='products/%Y/%m/%d',
                              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)
```

`django-parler` manages translations by generating another model for each translatable model. In the following schema, you can see the fields of the `Product` model and what the generated `ProductTranslation` model will look like:

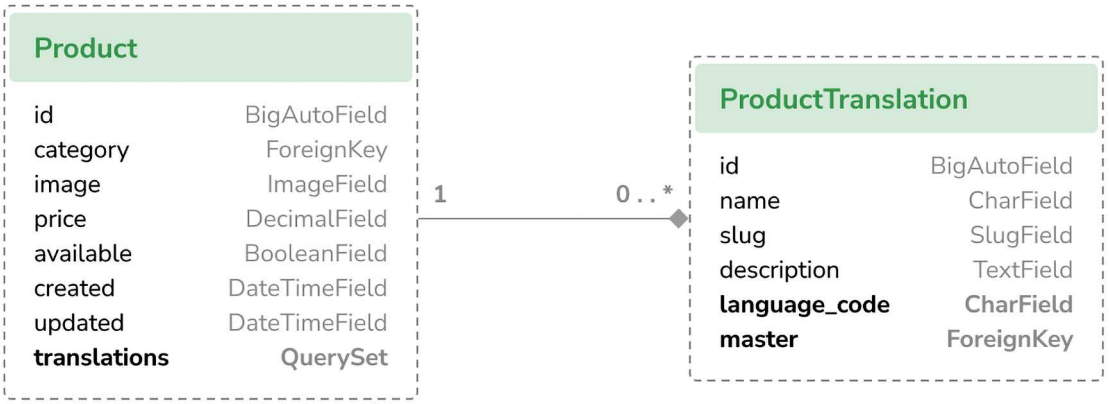


Figure 11.9: The Product model and related ProductTranslation model generated by django-parler

The `ProductTranslation` model generated by `django-parler` includes the `name`, `slug`, and `description` translatable fields, a `language_code` field, and a `ForeignKey` for the master `Product` object. There is a one-to-many relationship from `Product` to `ProductTranslation`. A `ProductTranslation` object will exist for each available language of each `Product` object.

Since Django uses a separate table for translations, there are some Django features that you can't use. It is not possible to use a default ordering by a translated field. You can filter by translated fields in queries, but you can't include a translatable field in the `ordering` `Meta` options. Also, you can't use indexes for the fields that are translated, as these fields will not exist in the original model, because they will reside in the translation model.

Edit the `models.py` file of the `shop` application and comment out the `ordering` and `indexes` attributes of the `Category` `Meta` class:

```
class Category(TranslatableModel):
    # ...
    class Meta:
        # ordering = ['name']
        # indexes = [
        #     models.Index(fields=['name']),
        # ]
        verbose_name = 'category'
        verbose_name_plural = 'categories'
```

You also have to comment out the `ordering` and attribute of the `Product Meta` class and the indexes that refer to the translated fields. Comment out the following lines of the `Product Meta` class:

```
class Product(TranslatableModel):
    # ...
    class Meta:
        # ordering = ['name']
        indexes = [
            # models.Index(fields=['id', 'slug']),
            # models.Index(fields=['name']),
            models.Index(fields=['-created']),
        ]
```

You can read more about the `django-parler` module's compatibility with Django at `https://django-parler.readthedocs.io/en/latest/compatibility.html`.

Integrating translations into the administration site

`django-parler` integrates smoothly with the Django administration site. It includes a `TranslatableAdmin` class that overrides the `ModelAdmin` class provided by Django to manage model translations.

Edit the `admin.py` file of the `shop` application and add the following import to it:

```
from parler.admin import TranslatableAdmin
```

Modify the `CategoryAdmin` and `ProductAdmin` classes to inherit from `TranslatableAdmin` instead of `ModelAdmin`. `django-parler` doesn't support the `prepopulated_fields` attribute, but it does support the `get_prepopulated_fields()` method that provides the same functionality. Let's change this accordingly. Edit the `admin.py` file to make it look like the following:

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

@admin.register(Category)
class CategoryAdmin(TranslatableAdmin):
    list_display = ['name', 'slug']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}

@admin.register(Product)
class ProductAdmin(TranslatableAdmin):
    list_display = ['name', 'slug', 'price',
                    'available', 'created', 'updated']
```

```
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'available']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}
```

You have adapted the administration site to work with the new translated models. You can now sync the database with the model changes that you made.

Creating migrations for model translations

Open the shell and run the following command to create a new migration for the model translations:

```
python manage.py makemigrations shop --name "translations"
```

You will see the following output:

```
Migrations for 'shop':
  shop/migrations/0002_translations.py
    - Create model CategoryTranslation
    - Create model ProductTranslation
    - Change Meta options on category
    - Change Meta options on product
    - Remove index shop_catego_name_289c7e_idx from category
    - Remove index shop_produc_id_f21274_idx from product
    - Remove index shop_produc_name_a2070e_idx from product
    - Remove field name from category
    - Remove field slug from category
    - Remove field description from product
    - Remove field name from product
    - Remove field slug from product
    - Add field master to producttranslation
    - Add field master to categorytranslation
    - Alter unique_together for producttranslation (1 constraint(s))
    - Alter unique_together for categorytranslation (1 constraint(s))
```

This migration automatically includes the `CategoryTranslation` and `ProductTranslation` models created dynamically by `django-parler`. It's important to note that this migration deletes the previous existing fields from your models. This means that you will lose that data and will need to set your categories and products again on the administration site after running it.

Edit the file `migrations/0002_translations.py` of the `shop` application and replace the two occurrences of the following line:

```
bases=(parler.models.TranslatedFieldsModelMixin, models.Model),
```

with the following one:

```
    bases=(parler.models.TranslatableModel, models.Model),
```

This is a fix for a minor issue found in the `django-parler` version you are using. This change is necessary to prevent the migration from failing when applying it. This issue is related to creating translations for existing fields in the model and will probably be fixed in newer `django-parler` versions.

Run the following command to apply the migration:

```
python manage.py migrate shop
```

You will see an output that ends with the following line:

```
Applying shop.0002_translations... OK
```

Your models are now synchronized with the database.

Run the development server using the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/en/admin/shop/category/` in your browser. You will see that existing categories lost their name and slug due to deleting those fields and using the translatable models generated by `django-parler` instead. You will just see a dash under each column like in *Figure 11.10*:

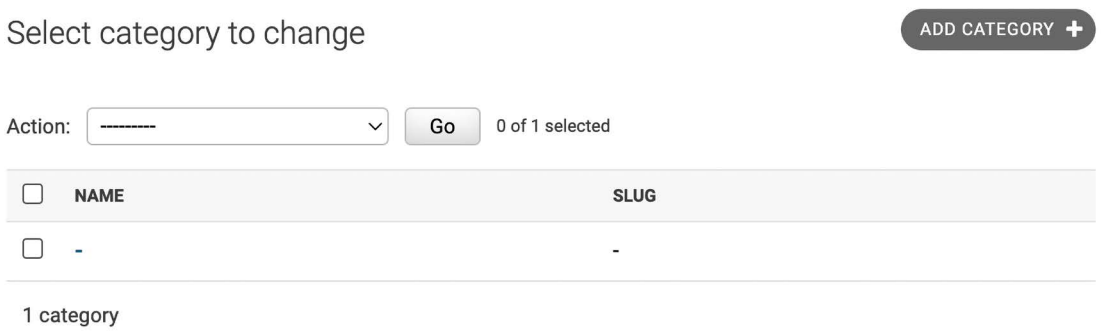


Figure 11.10: The category list on the Django administration site after creating the translation models

Click on the dash under the category name to edit it. You will see that the **Change category** page includes two different tabs, one for English and one for Spanish translations:

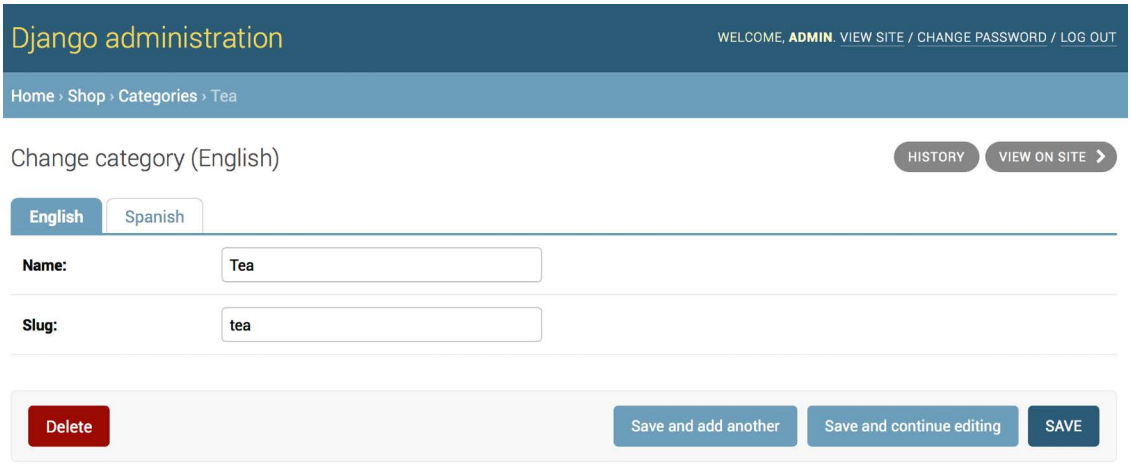


Figure 11.11: The category edit form, including the language tabs added by django-parler

Make sure that you fill in a name and slug for all existing categories. When you edit a category, enter the English details and click on **Save and continue editing**. Then, click on **Spanish**, add the Spanish translation for the fields, and click on **SAVE**:

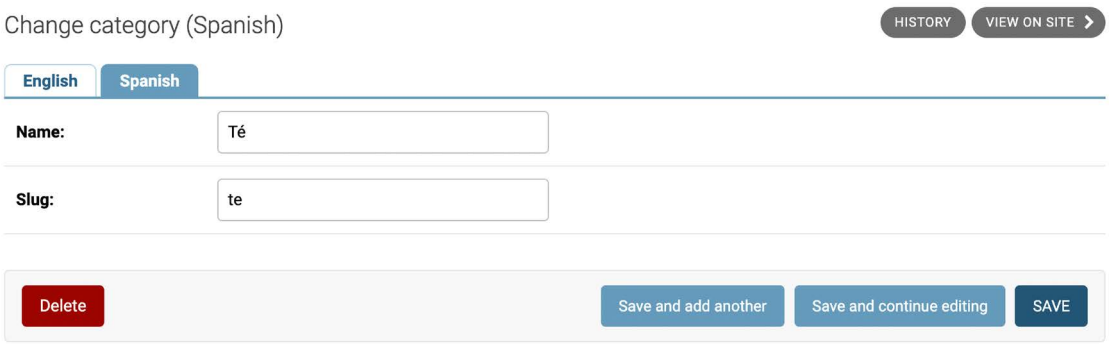


Figure 11.12: The Spanish translation of the category edit form

Make sure to save the changes before switching between the language tabs.

After completing the data for existing categories, open `http://127.0.0.1:8000/en/admin/shop/product/` and edit each of the products, providing an English and Spanish name, a slug, and a description.

Using translations with the ORM

You have to adapt your `shop` views to use translation QuerySets. Run the following command to open the Python shell:

```
python manage.py shell
```

Let's take a look at how you can retrieve and query translation fields. To get the object with translatable fields translated into a specific language, you can use Django's `activate()` function, as follows:

```
>>> from shop.models import Product
>>> from django.utils.translation import activate
>>> activate('es')
>>> product=Product.objects.first()
>>> product.name
'Té verde'
```

Another way to do this is by using the `language()` manager provided by `django-parler`, as follows:

```
>>> product=Product.objects.language('en').first()
>>> product.name
'Green tea'
```

When you access translated fields, they are resolved using the current language. You can set a different current language for an object to access that specific translation, as follows:

```
>>> product.set_current_language('es')
>>> product.name
'Té verde'
>>> product.get_current_language()
'es'
```

When performing a QuerySet using `filter()`, you can filter using the related translation objects with the `translations__` syntax, as follows:

```
>>> Product.objects.filter(translations__name='Green tea')
<TranslatableQuerySet [<Product: Té verde>]>
```

Adapting views for translations

Let's adapt the product catalog views. Edit the `views.py` file of the `shop` application and add the following code highlighted in bold to the `product_list` view:

```
def product_list(request, category_slug=None):
    category = None
    categories = Category.objects.all()
```

```
    products = Product.objects.filter(available=True)
    if category_slug:
        language = request.LANGUAGE_CODE
        category = get_object_or_404(Category,
                                     translations__language_code=language,
                                     translations__slug=category_slug)
        products = products.filter(category=category)
    return render(request,
                  'shop/product/list.html',
                  {'category': category,
                   'categories': categories,
                   'products': products})
```

Then, edit the `product_detail` view and add the following code highlighted in bold:

```
def product_detail(request, id, slug):
    language = request.LANGUAGE_CODE
    product = get_object_or_404(Product,
                                id=id,
                                translations__language_code=language,
                                translations__slug=slug,
                                available=True)
    cart_product_form = CartAddProductForm()
    r = Recommender()
    recommended_products = r.suggest_products_for([product], 4)
    return render(request,
                  'shop/product/detail.html',
                  {'product': product,
                   'cart_product_form': cart_product_form,
                   'recommended_products': recommended_products})
```

The `product_list` and `product_detail` views are now adapted to retrieve objects using translated fields.

Run the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/es/` in your browser. You should see the product list page, including all products translated into Spanish:

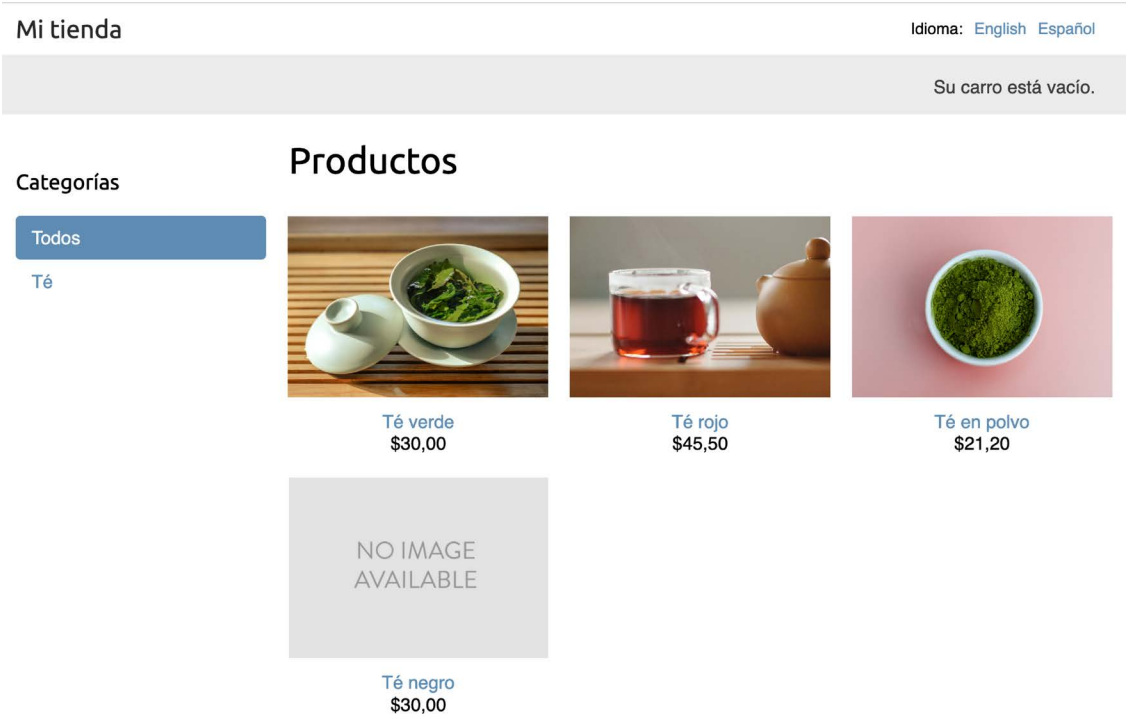


Figure 11.13: The Spanish version of the product list page

Now, each product's URL is built using the `slug` field translated into the current language. For example, the URL for a product in Spanish is `http://127.0.0.1:8000/es/2/te-rojo/`, whereas, in English, the URL is `http://127.0.0.1:8000/en/2/red-tea/`. If you navigate to a product details page, you will see the translated URL and the contents of the selected language, as shown in the following example:

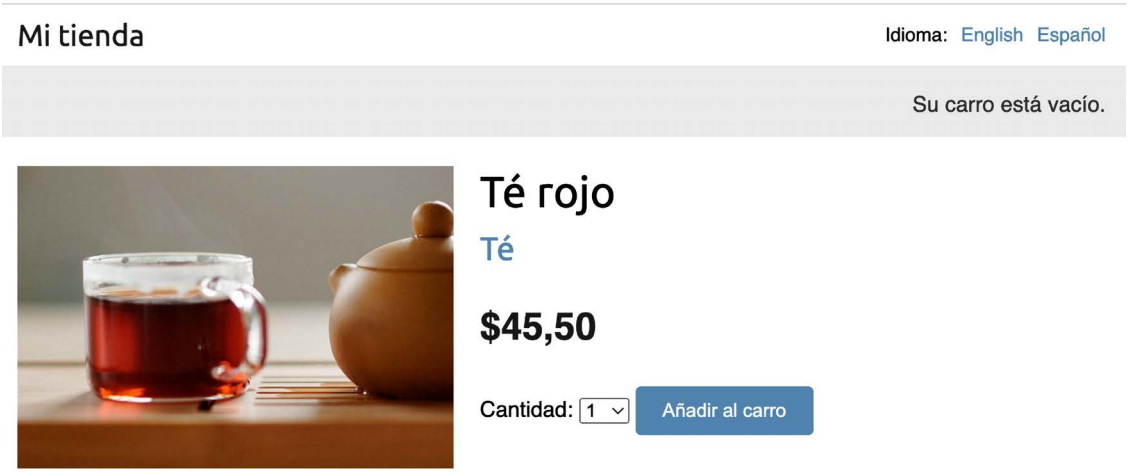


Figure 11.14: The Spanish version of the product details page

If you want to know more about `django-parler`, you can find the full documentation at `https://django-parler.readthedocs.io/en/latest/`.

You have learned how to translate Python code, templates, URL patterns, and model fields. To complete the internationalization and localization process, you need to use localized formatting for dates, times, and numbers as well.

Format localization

Depending on the user's locale, you might want to display dates, times, and numbers in different formats. Localized formatting can be activated by changing the `USE_L10N` setting to `True` in the `settings.py` file of your project.

When `USE_L10N` is enabled, Django will try to use a locale-specific format whenever it outputs a value in a template. You can see that decimal numbers in the English version of your site are displayed with a dot separator for decimal places, while in the Spanish version, they are displayed using a comma. This is due to the locale formats specified for the `es` locale by Django. You can take a look at the Spanish formatting configuration at `https://github.com/django/django/blob/stable/4.0.x/django/conf/locale/es/formats.py`.

Normally, you will set the `USE_L10N` setting to `True` and let Django apply the format localization for each locale. However, there might be situations in which you don't want to use localized values. This is especially relevant when outputting JavaScript or JSON, which has to provide a machine-readable format.

Django offers a `{% localize %}` template tag that allows you to turn on/off localization for template fragments. This gives you control over localized formatting. You will have to load the `l10n` tags to be able to use this template tag. The following is an example of how to turn localization on and off in a template:

```
{% load l10n %}

{% localize on %}
  {{ value }}
{% endlocalize %}

{% localize off %}
  {{ value }}
{% endlocalize %}
```

Django also offers the `localize` and `unlocalize` template filters to force or avoid the localization of a value. These filters can be applied as follows:

```
{{ value|localize }}
{{ value|unlocalize }}
```

You can also create custom format files to specify locale formatting. You can find further information about format localization at `https://docs.djangoproject.com/en/4.1/topics/i18n/formatting/`.

Next, you will learn how to create localized form fields.

Using django-localflavor to validate form fields

`django-localflavor` is a third-party module that contains a collection of utilities, such as form fields or model fields, that are specific for each country. It's very useful for validating local regions, local phone numbers, identity card numbers, social security numbers, and so on. The package is organized into a series of modules named after ISO 3166 country codes.

Install `django-localflavor` using the following command:

```
pip install django-localflavor==3.1
```

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

```
INSTALLED_APPS = [
    # ...
    'localflavor',
]
```

You are going to add the United States zip code field so that a valid United States zip code is required to create a new order.

Edit the `forms.py` file of the `orders` application and make it look like the following:

```
from django import forms
from localflavor.us.forms import USZipCodeField
from .models import Order

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

You import the `USZipCodeField` field from the `us` package of `localflavor` and use it for the `postal_code` field of the `OrderCreateForm` form.

Run the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/en/orders/create/` in your browser. Fill in all the fields, enter a three-letter zip code, and then submit the form. You will get the following validation error, which is raised by `USZipCodeField`:

```
Enter a zip code in the format XXXXX or XXXXX-XXXX.
```

Figure 11.15 shows the form validation error:

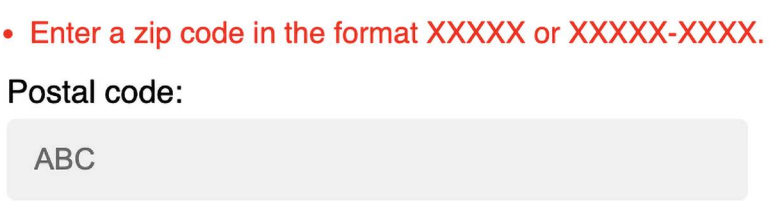


Figure 11.15: The validation error for an invalid US zip code

This is just a brief example of how to use a custom field from `localflavor` in your own project for validation purposes. The local components provided by `localflavor` are very useful for adapting your application to specific countries. You can read the `django-localflavor` documentation and see all the available local components for each country at `https://django-localflavor.readthedocs.io/en/latest/`.

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-4-by-example/tree/main/Chapter11`
- List of valid language IDs – `http://www.i18nguy.com/unicode/language-identifiers.html`
- List of internationalization and localization settings – `https://docs.djangoproject.com/en/4.1/ref/settings/#globalization-i18n-l10n`
- Homebrew package manager – `https://brew.sh/`
- Installing gettext on Windows – `https://docs.djangoproject.com/en/4.1/topics/i18n/translation/#gettext-on-windows`
- Precompiled gettext binary installer for Windows – `https://mlocati.github.io/articles/gettext-iconv-windows.html`
- Documentation about translations – `https://docs.djangoproject.com/en/4.1/topics/i18n/translation/`
- Poedit translation file editor – `https://poedit.net/`
- Documentation for Django Rosetta – `https://django-rosetta.readthedocs.io/`
- The `django-parler` module's compatibility with Django – `https://django-parler.readthedocs.io/en/latest/compatibility.html`
- Documentation for `django-parler` – `https://django-parler.readthedocs.io/en/latest/`
- Django formatting configuration for the Spanish locale – `https://github.com/django/django/blob/stable/4.0.x/django/conf/locale/es/formats.py`
- Django format localization – `https://docs.djangoproject.com/en/4.1/topics/i18n/formatting/`
- Documentation for `django-localflavor` – `https://django-localflavor.readthedocs.io/en/latest/`

Summary

In this chapter, you learned the basics of the internationalization and localization of Django projects. You marked code and template strings for translation, and you discovered how to generate and compile translation files. You also installed Rosetta in your project to manage translations through a web interface. You translated URL patterns, and you created a language selector to allow users to switch the language of the site. Then, you used `django-parler` to translate models, and you used `django-localflavor` to validate localized form fields.

In the next chapter, you will start a new Django project that will consist of an e-learning platform. You will create the application models, and you will learn how to create and apply fixtures to provide initial data for the models. You will build a custom model field and use it in your models. You will also build authentication views for your new application.

12

Building an E-Learning Platform

In the previous chapter, you learned the basics of the internationalization and localization of Django projects. You added internationalization to your online shop project. You learned how to translate Python strings, templates, and models. You also learned how to manage translations, and you created a language selector and added localized fields to your forms.

In this chapter, you will start a new Django project that will consist of an e-learning platform with your own **content management system** (**CMS**). Online learning platforms are a great example of applications where you need to provide tools to generate content with flexibility in mind.

In this chapter, you will learn how to:

- Create models for the CMS
- Create fixtures for your models and apply them
- Use model inheritance to create data models for polymorphic content
- Create custom model fields
- Order course contents and modules
- Build authentication views for the CMS

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter12`.

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

Setting up the e-learning project

Your final practical project will be an e-learning platform. First, create a virtual environment for your new project within the `env/` directory with the following command:

```
python -m venv env/educa
```

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

```
source env/educa/bin/activate
```

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

```
.\env\educa\Scripts\activate
```

Install Django in your virtual environment with the following command:

```
pip install Django~=4.1.0
```

You are going to manage image uploads in your project, so you also need to install `Pillow` with the following command:

```
pip install Pillow==9.2.0
```

Create a new project using the following command:

```
django-admin startproject educa
```

Enter the new `educa` directory and create a new application using the following commands:

```
cd educa
django-admin startapp courses
```

Edit the `settings.py` file of the `educa` project and add `courses` to the `INSTALLED_APPS` setting, as follows. The new line is highlighted in bold:

```
INSTALLED_APPS = [
    'courses.apps.CoursesConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

The `courses` application is now active for the project. Next, we are going to prepare our project to serve media files, and we will define the models for the courses and course contents.

Serving media files

Before creating the models for courses and course contents, we will prepare the project to serve media files. Course instructors will be able to upload media files to course contents using the CMS that we will build. Therefore, we will configure the project to serve media files.

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

```
MEDIA_URL = 'media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

This will enable Django to manage file uploads and serve media files. `MEDIA_URL` is the base URL used to serve the media files uploaded by users. `MEDIA_ROOT` is the local path where they reside. Paths and URLs for files are built dynamically by prepending the project path or the media URL to them for portability.

Now, edit the main `urls.py` file of the `educa` project and modify the code, as follows. New lines are highlighted in bold:

```
from django.contrib import admin
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
]

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

We have added the `static()` helper function to serve media files with the Django development server during development (that is, when the `DEBUG` setting is set to `True`).

Remember that the `static()` helper function is suitable for development but not for production use. Django is very inefficient at serving static files. Never serve your static files with Django in a production environment. You will learn how to serve static files in a production environment in *Chapter 17*, *Going Live*.

The project is now ready to serve media files. Let's create the models for the courses and course contents.

Building the course models

Your e-learning platform will offer courses on various subjects. Each course will be divided into a configurable number of modules, and each module will contain a configurable number of contents. The contents will be of various types: text, files, images, or videos. The following example shows what the data structure of your course catalog will look like:

```
Subject 1
  Course 1
    Module 1
```

```
      Content 1 (image)
      Content 2 (text)
    Module 2
      Content 3 (text)
      Content 4 (file)
      Content 5 (video)
      ...
```

Let's build the course models. Edit the `models.py` file of the `courses` application and add the following code to it:

```
from django.db import models
from django.contrib.auth.models import User

class Subject(models.Model):
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)

    class Meta:
        ordering = ['title']

    def __str__(self):
        return self.title


class Course(models.Model):
    owner = models.ForeignKey(User,
                              related_name='courses_created',
                              on_delete=models.CASCADE)
    subject = models.ForeignKey(Subject,
                                related_name='courses',
                                on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)
    overview = models.TextField()
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ['-created']

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

```
class Module(models.Model):
    course = models.ForeignKey(Course,
                               related_name='modules',
                               on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)

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

These are the initial `Subject`, `Course`, and `Module` models. The `Course` model fields are as follows:

- `owner`: The instructor who created this course.
- `subject`: The subject that this course belongs to. It is a `ForeignKey` field that points to the `Subject` model.
- `title`: The title of the course.
- `slug`: The slug of the course. This will be used in URLs later.
- `overview`: A `TextField` column to store an overview of the course.
- `created`: The date and time when the course was created. It will be automatically set by Django when creating new objects because of `auto_now_add=True`.

Each course is divided into several modules. Therefore, the `Module` model contains a `ForeignKey` field that points to the `Course` model.

Open the shell and run the following command to create the initial migration for this application:

```
python manage.py makemigrations
```

You will see the following output:

```
Migrations for 'courses':
  courses/migrations/0001_initial.py:
    - Create model Course
    - Create model Module
    - Create model Subject
    - Add field subject to course
```

Then, run the following command to apply all migrations to the database:

```
python manage.py migrate
```

You should see output that includes all applied migrations, including those of Django. The output will contain the following line:

```
Applying courses.0001_initial... OK
```

The models of your `courses` application have been synced with the database.

Registering the models in the administration site

Let's add the course models to the administration site. Edit the `admin.py` file inside the `courses` application directory and add the following code to it:

```
from django.contrib import admin
from .models import Subject, Course, Module

@admin.register(Subject)
class SubjectAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug']
    prepopulated_fields = {'slug': ('title',)}

class ModuleInline(admin.StackedInline):
    model = Module

@admin.register(Course)
class CourseAdmin(admin.ModelAdmin):
    list_display = ['title', 'subject', 'created']
    list_filter = ['created', 'subject']
    search_fields = ['title', 'overview']
    prepopulated_fields = {'slug': ('title',)}
    inlines = [ModuleInline]
```

The models for the course application are now registered on the administration site. Remember that you use the `@admin.register()` decorator to register models on the administration site.

Using fixtures to provide initial data for models

Sometimes, you might want to prepopulate your database with hardcoded data. This is useful for automatically including initial data in the project setup, instead of having to add it manually. Django comes with a simple way to load and dump data from the database into files that are called **fixtures**. Django supports fixtures in JSON, XML, or YAML formats. You are going to create a fixture to include several initial `Subject` objects for your project.

First, create a superuser using the following command:

```
python manage.py createsuperuser
```

Then, run the development server using the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/courses/subject/` in your browser. Create several subjects using the administration site. The change list page should look as follows:

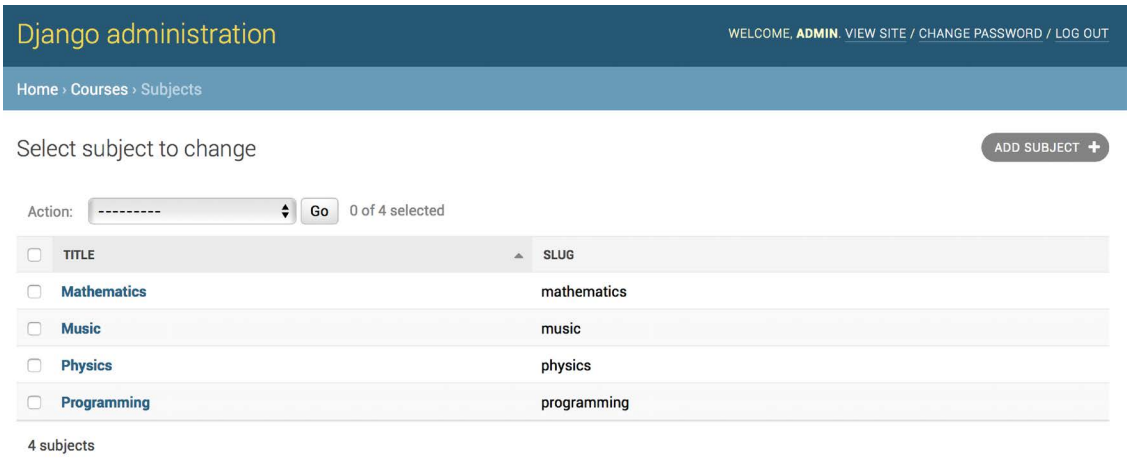


Figure 12.1: The subject change list view on the administration site

Run the following command from the shell:

```
python manage.py dumpdata courses --indent=2
```

You will see an output similar to the following:

```
[
{
  "model": "courses.subject",
  "pk": 1,
  "fields": {
    "title": "Mathematics",
    "slug": "mathematics"
  }
},
{
  "model": "courses.subject",
  "pk": 2,
  "fields": {
    "title": "Music",
    "slug": "music"
  }
```

```
},
{
  "model": "courses.subject",
  "pk": 3,
  "fields": {
    "title": "Physics",
    "slug": "physics"
  }
},
{
  "model": "courses.subject",
  "pk": 4,
  "fields": {
    "title": "Programming",
    "slug": "programming"
  }
}
]
```

The `dumpdata` command dumps data from the database into the standard output, serialized in JSON format by default. The resulting data structure includes information about the model and its fields for Django to be able to load it into the database.

You can limit the output to the models of an application by providing the application names to the command, or specifying single models for outputting data using the `app.Model` format. You can also specify the format using the `--format` flag. By default, `dumpdata` outputs the serialized data to the standard output. However, you can indicate an output file using the `--output` flag. The `--indent` flag allows you to specify indentations. For more information on `dumpdata` parameters, run `python manage.py dumpdata --help`.

Save this dump to a fixtures file in a new `fixtures/` directory in the `courses` application using the following commands:

```
mkdir courses/fixtures
python manage.py dumpdata courses --indent=2 --output=courses/fixtures/
subjects.json
```

Run the development server and use the administration site to remove the subjects you created, as shown in *Figure 12.2*:

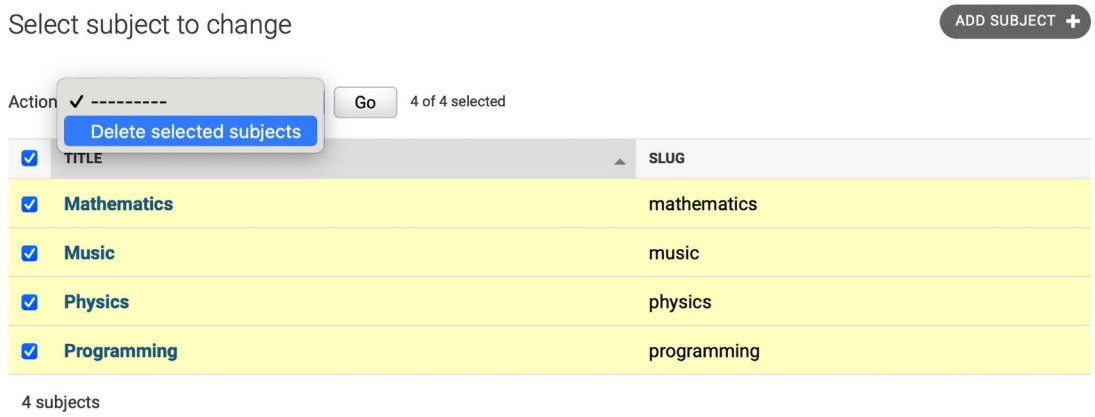


Figure 12.2: Deleting all existing subjects

After deleting all subjects, load the fixture into the database using the following command:

```
python manage.py loaddata subjects.json
```

All `Subject` objects included in the fixture are loaded into the database again:

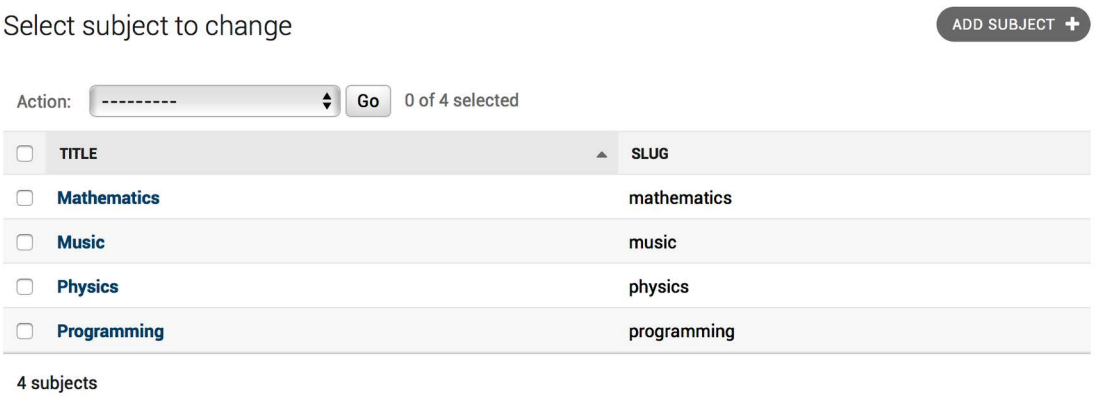


Figure 12.3: Subjects from the fixture are now loaded into the database

By default, Django looks for files in the `fixtures/` directory of each application, but you can specify the complete path to the fixture file for the `loaddata` command. You can also use the `FIXTURE_DIRS` setting to tell Django additional directories to look in for fixtures.

Fixtures are not only useful for setting up initial data, but also for providing sample data for your application or data required for your tests.

You can read about how to use fixtures for testing at `https://docs.djangoproject.com/en/4.1/topics/testing/tools/#fixture-loading`.

If you want to load fixtures in model migrations, look at Django's documentation about data migrations. You can find the documentation for migrating data at `https://docs.djangoproject.com/en/4.1/topics/migrations/#data-migrations`.

You have created the models to manage course subjects, courses, and course modules. Next, you will create models to manage different types of module contents.

Creating models for polymorphic content

You plan to add different types of content to the course modules, such as text, images, files, and videos. **Polymorphism** is the provision of a single interface to entities of different types. You need a versatile data model that allows you to store diverse content that is accessible through a single interface. In *Chapter 7*, *Tracking User Actions*, you learned about the convenience of using generic relations to create foreign keys that can point to the objects of any model. You are going to create a `Content` model that represents the modules' contents and define a generic relation to associate any object with the content object.

Edit the `models.py` file of the `courses` application and add the following imports:

```
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey
```

Then, add the following code to the end of the file:

```
class Content(models.Model):
    module = models.ForeignKey(Module,
                               related_name='contents',
                               on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType,
                               on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    item = GenericForeignKey('content_type', 'object_id')
```

This is the `Content` model. A module contains multiple contents, so you define a `ForeignKey` field that points to the `Module` model. You can also set up a generic relation to associate objects from different models that represent different types of content. Remember that you need three different fields to set up a generic relation. In your `Content` model, these are:

- `content_type`: A `ForeignKey` field to the `ContentType` model.
- `object_id`: A `PositiveIntegerField` to store the primary key of the related object.
- `item`: A `GenericForeignKey` field to the related object combining the two previous fields.

Only the `content_type` and `object_id` fields have a corresponding column in the database table of this model. The `item` field allows you to retrieve or set the related object directly, and its functionality is built on top of the other two fields.

You are going to use a different model for each type of content. Your `Content` models will have some common fields, but they will differ in the actual data they can store. This is how you will create a single interface for different types of content.

Using model inheritance

Django supports model inheritance. It works in a similar way to standard class inheritance in Python. Django offers the following three options to use model inheritance:

- **Abstract models**: Useful when you want to put some common information into several models.
- **Multi-table model inheritance**: Applicable when each model in the hierarchy is considered a complete model by itself.
- **Proxy models**: Useful when you need to change the behavior of a model, for example, by including additional methods, changing the default manager, or using different meta options.

Let's take a closer look at each of them.

Abstract models

An abstract model is a base class in which you define the fields you want to include in all child models. Django doesn't create any database tables for abstract models. A database table is created for each child model, including the fields inherited from the abstract class and the ones defined in the child model.

To mark a model as abstract, you need to include `abstract=True` in its `Meta` class. Django will recognize that it is an abstract model and will not create a database table for it. To create child models, you just need to subclass the abstract model.

The following example shows an abstract `Content` model and a child `Text` model:

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)
    class Meta:
```

```
        abstract = True

class Text(BaseContent):
    body = models.TextField()
```

In this case, Django would create a table for the `Text` model only, including the `title`, `created`, and `body` fields.

Multi-table model inheritance

In multi-table inheritance, each model corresponds to a database table. Django creates a `OneToOneField` field for the relationship between the child model and its parent model. To use multi-table inheritance, you have to subclass an existing model. Django will create a database table for both the original model and the sub-model. The following example shows multi-table inheritance:

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class Text(BaseContent):
    body = models.TextField()
```

Django will include an automatically generated `OneToOneField` field in the `Text` model and create a database table for each model.

Proxy models

A proxy model changes the behavior of a model. Both models operate on the database table of the original model. To create a proxy model, add `proxy=True` to the `Meta` class of the model. The following example illustrates how to create a proxy model:

```
from django.db import models
from django.utils import timezone

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class OrderedContent(BaseContent):
    class Meta:
        proxy = True
        ordering = ['created']
```

```
    def created_delta(self):
        return timezone.now() - self.created
```

Here, you define an `OrderedContent` model that is a proxy model for the `Content` model. This model provides a default ordering for QuerySets and an additional `created_delta()` method. Both models, `Content` and `OrderedContent`, operate on the same database table, and objects are accessible via the ORM through either model.

Creating the Content models

The `Content` model of your `courses` application contains a generic relation to associate different types of content with it. You will create a different model for each type of content. All `Content` models will have some fields in common and additional fields to store custom data. You are going to create an abstract model that provides the common fields for all `Content` models.

Edit the `models.py` file of the `courses` application and add the following code to it:

```
class ItemBase(models.Model):
    owner = models.ForeignKey(User,
                        related_name='%(class)s_related',
                        on_delete=models.CASCADE)
    title = models.CharField(max_length=250)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True

    def __str__(self):
        return self.title

class Text(ItemBase):
    content = models.TextField()

class File(ItemBase):
    file = models.FileField(upload_to='files')

class Image(ItemBase):
       file = models.FileField(upload_to='images')

class Video(ItemBase):
    url = models.URLField()
```

In this code, you define an abstract model named `ItemBase`. Therefore, you set `abstract=True` in its `Meta` class.

In this model, you define the `owner`, `title`, `created`, and `updated` fields. These common fields will be used for all types of content.

The `owner` field allows you to store which user created the content. Since this field is defined in an abstract class, you need a different `related_name` for each sub-model. Django allows you to specify a placeholder for the model class name in the `related_name` attribute as `%(class)s`. By doing so, the `related_name` for each child model will be generated automatically. Since you are using `'%(class)s_related'` as the `related_name`, the reverse relationship for child models will be `text_related`, `file_related`, `image_related`, and `video_related`, respectively.

You have defined four different `Content` models that inherit from the `ItemBase` abstract model. They are as follows:

- `Text`: To store text content
- `File`: To store files, such as PDFs
- `Image`: To store image files
- `Video`: To store videos; you use an `URLField` field to provide a video URL in order to embed it

Each child model contains the fields defined in the `ItemBase` class in addition to its own fields. A database table will be created for the `Text`, `File`, `Image`, and `Video` models, respectively. There will be no database table associated with the `ItemBase` model since it is an abstract model.

Edit the `Content` model you created previously and modify its `content_type` field, as follows:

```
content_type = models.ForeignKey(ContentType,
                  on_delete=models.CASCADE,
                  limit_choices_to={'model__in':(
                                      'text',
                                      'video',
                                      'image',
                                      'file')})
```

You add a `limit_choices_to` argument to limit the `ContentType` objects that can be used for the generic relation. You use the `model__in` field lookup to filter the query to the `ContentType` objects with a `model` attribute that is `'text'`, `'video'`, `'image'`, or `'file'`.

Let's create a migration to include the new models you have added. Run the following command from the command line:

```
python manage.py makemigrations
```

You will see the following output:

```
Migrations for 'courses':
  courses/migrations/0002_video_text_image_file_content.py
```

```
    - Create model Video
    - Create model Text
    - Create model Image
    - Create model File
    - Create model Content
```

Then, run the following command to apply the new migration:

```
python manage.py migrate
```

The output you see should end with the following line:

```
Applying courses.0002_video_text_image_file_content... OK
```

You have created models that are suitable for adding diverse content to the course modules. However, there is still something missing in your models: the course modules and contents should follow a particular order. You need a field that allows you to order them easily.

Creating custom model fields

Django comes with a complete collection of model fields that you can use to build your models. However, you can also create your own model fields to store custom data or alter the behavior of existing fields.

You need a field that allows you to define an order for the objects. An easy way to specify an order for objects using existing Django fields is by adding a `PositiveIntegerField` to your models. Using integers, you can easily specify the order of the objects. You can create a custom order field that inherits from `PositiveIntegerField` and provides additional behavior.

There are two relevant functionalities that you will build into your order field:

- **Automatically assign an order value when no specific order is provided**: When saving a new object with no specific order, your field should automatically assign the number that comes after the last existing ordered object. If there are two objects with orders `1` and `2` respectively, when saving a third object, you should automatically assign order `3` to it if no specific order has been provided.
- **Order objects with respect to other fields**: Course modules will be ordered with respect to the course they belong to and module contents with respect to the module they belong to.

Create a new `fields.py` file inside the `courses` application directory and add the following code to it:

```
from django.db import models
from django.core.exceptions import ObjectDoesNotExist

class OrderField(models.PositiveIntegerField):
    def __init__(self, for_fields=None, *args, **kwargs):
        self.for_fields = for_fields
        super().__init__(*args, **kwargs)
```

```
    def pre_save(self, model_instance, add):
        if getattr(model_instance, self.attname) is None:
            # no current value
            try:
                qs = self.model.objects.all()
                if self.for_fields:
                    # filter by objects with the same field values
                    # for the fields in "for_fields"
                    query = {field: getattr(model_instance, field)\
                    for field in self.for_fields}
                    qs = qs.filter(**query)
                # get the order of the last item
                last_item = qs.latest(self.attname)
                value = last_item.order + 1
            except ObjectDoesNotExist:
                value = 0
            setattr(model_instance, self.attname, value)
            return value
        else:
            return super().pre_save(model_instance, add)
```

This is the custom `OrderField`. It inherits from the `PositiveIntegerField` field provided by Django. Your `OrderField` field takes an optional `for_fields` parameter, which allows you to indicate the fields used to order the data.

Your field overrides the `pre_save()` method of the `PositiveIntegerField` field, which is executed before saving the field to the database. In this method, you perform the following actions:

1. You check whether a value already exists for this field in the model instance. You use `self.attname`, which is the attribute name given to the field in the model. If the attribute's value is different from `None`, you calculate the order you should give it as follows:
    1. You build a QuerySet to retrieve all objects for the field's model. You retrieve the model class the field belongs to by accessing `self.model`.
    2. If there are any field names in the `for_fields` attribute of the field, you filter the QuerySet by the current value of the model fields in `for_fields`. By doing so, you calculate the order with respect to the given fields.
    3. You retrieve the object with the highest order with `last_item = qs.latest(self.attname)` from the database. If no object is found, you assume this object is the first one and assign order `0` to it.
    4. If an object is found, you add `1` to the highest order found.
    5. You assign the calculated order to the field's value in the model instance using `setattr()` and return it.

2. If the model instance has a value for the current field, you use it instead of calculating it.

When you create custom model fields, make them generic. Avoid hardcoding data that depends on a specific model or field. Your field should work in any model.

You can find more information about writing custom model fields at `https://docs.djangoproject.com/en/4.1/howto/custom-model-fields/`.

Adding ordering to module and content objects

Let's add the new field to your models. Edit the `models.py` file of the `courses` application, and import the `OrderField` class and a field to the `Module` model, as follows:

```
from .fields import OrderField

class Module(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['course'])
```

You name the new field `order` and specify that the ordering is calculated with respect to the course by setting `for_fields=['course']`. This means that the order for a new module will be assigned by adding `1` to the last module of the same `Course` object.

Now, you can edit the `__str__()` method of the `Module` model to include its order, as follows:

```
class Module(models.Model):
    # ...
    def __str__(self):
        return f'{self.order}. {self.title}'
```

Module contents also need to follow a particular order. Add an `OrderField` field to the `Content` model, as follows:

```
class Content(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['module'])
```

This time, you specify that the order is calculated with respect to the `module` field.

Finally, let's add a default ordering for both models. Add the following `Meta` class to the `Module` and `Content` models:

```
class Module(models.Model):
    # ...
    class Meta:
        ordering = ['order']
```

```
class Content(models.Model):
    # ...
    class Meta:
        ordering = ['order']
```

The `Module` and `Content` models should now look as follows:

```
class Module(models.Model):
    course = models.ForeignKey(Course,
                               related_name='modules',
                               on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    order = OrderField(blank=True, for_fields=['course'])

    class Meta:
        ordering = ['order']

    def __str__(self):
        return f'{self.order}. {self.title}'


class Content(models.Model):
    module = models.ForeignKey(Module,
                               related_name='contents',
                               on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType,
                            on_delete=models.CASCADE,
                            limit_choices_to={'model__in':(
                                        'text',
                                        'video',
                                        'image',
                                        'file')})
    object_id = models.PositiveIntegerField()
    item = GenericForeignKey('content_type', 'object_id')
    order = OrderField(blank=True, for_fields=['module'])

    class Meta:
            ordering = ['order']
```

Let's create a new model migration that reflects the new order fields. Open the shell and run the following command:

```
python manage.py makemigrations courses
```

You will see the following output:

```
It is impossible to add a non-nullable field 'order' to content without
specifying a default. This is because the database needs something to populate
existing rows.
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null
value for this column)
 2) Quit and manually define a default value in models.py.
Select an option:
```

Django is telling you that you have to provide a default value for the new `order` field for existing rows in the database. If the field includes `null=True`, it accepts null values and Django creates the migration automatically instead of asking for a default value. You can specify a default value, or cancel the migration and add a `default` attribute to the `order` field in the `models.py` file before creating the migration.

Enter `1` and press *Enter* to provide a default value for existing records. You will see the following output:

```
Please enter the default value as valid Python.
The datetime and django.utils.timezone modules are available, so it is possible
to provide e.g. timezone.now as a value.
Type 'exit' to exit this prompt
>>>
```

Enter `0` so that this is the default value for existing records and press *Enter*. Django will ask you for a default value for the `Module` model too. Choose the first option and enter `0` as the default value again. Finally, you will see an output similar to the following one:

```
Migrations for 'courses':
courses/migrations/0003_alter_content_options_alter_module_options_and_more.py
    - Change Meta options on content
    - Change Meta options on module
    - Add field order to content
    - Add field order to module
```

Then, apply the new migrations with the following command:

```
python manage.py migrate
```

The output of the command will inform you that the migration was successfully applied, as follows:

```
Applying courses.0003_alter_content_options_alter_module_options_and_more... OK
```

Let's test your new field. Open the shell with the following command:

```
python manage.py shell
```

Create a new course, as follows:

```
>>> from django.contrib.auth.models import User
>>> from courses.models import Subject, Course, Module
>>> user = User.objects.last()
>>> subject = Subject.objects.last()
>>> c1 = Course.objects.create(subject=subject, owner=user, title='Course 1',
slug='course1')
```

You have created a course in the database. Now, you will add modules to the course and see how their order is automatically calculated. You create an initial module and check its order:

```
>>> m1 = Module.objects.create(course=c1, title='Module 1')
>>> m1.order
0
```

`OrderField` sets its value to `0`, since this is the first `Module` object created for the given course. You can create a second module for the same course:

```
>>> m2 = Module.objects.create(course=c1, title='Module 2')
>>> m2.order
1
```

`OrderField` calculates the next order value, adding `1` to the highest order for existing objects. Let's create a third module, forcing a specific order:

```
>>> m3 = Module.objects.create(course=c1, title='Module 3', order=5)
>>> m3.order
5
```

If you provide a custom order when creating or saving an object, `OrderField` will use that value instead of calculating the order.

Let's add a fourth module:

```
>>> m4 = Module.objects.create(course=c1, title='Module 4')
>>> m4.order
6
```

The order for this module has been automatically set. Your `OrderField` field does not guarantee that all order values are consecutive. However, it respects existing order values and always assigns the next order based on the highest existing order.

Let's create a second course and add a module to it:

```
>>> c2 = Course.objects.create(subject=subject, title='Course 2',
slug='course2', owner=user)
>>> m5 = Module.objects.create(course=c2, title='Module 1')
>>> m5.order
0
```

To calculate the new module's order, the field only takes into consideration existing modules that belong to the same course. Since this is the first module of the second course, the resulting order is `0`. This is because you specified `for_fields=['course']` in the `order` field of the `Module` model.

Congratulations! You have successfully created your first custom model field. Next, you are going to create an authentication system for the CMS.

Adding authentication views

Now that you have created a polymorphic data model, you are going to build a CMS to manage the courses and their contents. The first step is to add an authentication system for the CMS.

Adding an authentication system

You are going to use Django's authentication framework for users to authenticate to the e-learning platform. Both instructors and students will be instances of Django's `User` model, so they will be able to log in to the site using the authentication views of `django.contrib.auth`.

Edit the main `urls.py` file of the `educa` project and include the `login` and `logout` views of Django's authentication framework:

```
from django.contrib import admin
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('accounts/login/', auth_views.LoginView.as_view(),
         name='login'),
    path('accounts/logout/', auth_views.LogoutView.as_view(),
         name='logout'),
    path('admin/', admin.site.urls),
]

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

Creating the authentication templates

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

```
templates/
    base.html
    registration/
        login.html
        logged_out.html
```

Before building the authentication templates, you need to prepare the base template for your project. Edit the `base.html` template file and add the following content to it:

```
{% load static %}
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>{% block title %}Educa{% endblock %}</title>
    <link href="{% static "css/base.css" %}" rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <a href="/" class="logo">Educa</a>
      <ul class="menu">
        {% if request.user.is_authenticated %}
          <li><a href="{% url "logout" %}">Sign out</a></li>
        {% else %}
          <li><a href="{% url "login" %}">Sign in</a></li>
        {% endif %}
      </ul>
    </div>
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
    <script>
      document.addEventListener('DOMContentLoaded', (event) => {
        // DOM loaded
        {% block domready %}
        {% endblock %}
      })
    </script>
```

```
    </body>
</html>
```

This is the base template that will be extended by the rest of the templates. In this template, you define the following blocks:

- `title`: The block for other templates to add a custom title for each page.
- `content`: The main block for content. All templates that extend the base template should add content to this block.
- `domready`: Located inside the JavaScript event listener for the `DOMContentLoaded` event. It allows you to execute code when the **Document Object Model** (**DOM**) has finished loading.

The CSS styles used in this template are located in the `static/` directory of the `courses` application in the code that comes with this chapter. Copy the `static/` directory into the same directory of your project to use them. You can find the contents of the directory at `https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter12/educa/courses/static`.

Edit the `registration/login.html` template and add the following code to it:

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

{% block title %}Log-in{% endblock %}

{% block content %}
  <h1>Log-in</h1>
  <div class="module">
    {% if form.errors %}
      <p>Your username and password didn't match. Please try again.</p>
    {% else %}
      <p>Please, use the following form to log-in:</p>
    {% endif %}
    <div class="login-form">
      <form action="{% url 'login' %}" method="post">
        {{ form.as_p }}
        {% csrf_token %}
        <input type="hidden" name="next" value="{{ next }}" />
        <p><input type="submit" value="Log-in"></p>
      </form>
    </div>
  </div>
{% endblock %}
```

This is a standard login template for Django's `login` view.

Edit the `registration/logged_out.html` template and add the following code to it:

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

{% block title %}Logged out{% endblock %}

{% block content %}
  <h1>Logged out</h1>
  <div class="module">
    <p>
      You have been successfully logged out.
      You can <a href="{% url "login" %}">log-in again</a>.
    </p>
  </div>
{% endblock %}
```

This is the template that will be displayed to the user after logging out. Run the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/accounts/login/` in your browser. You should see the login page:

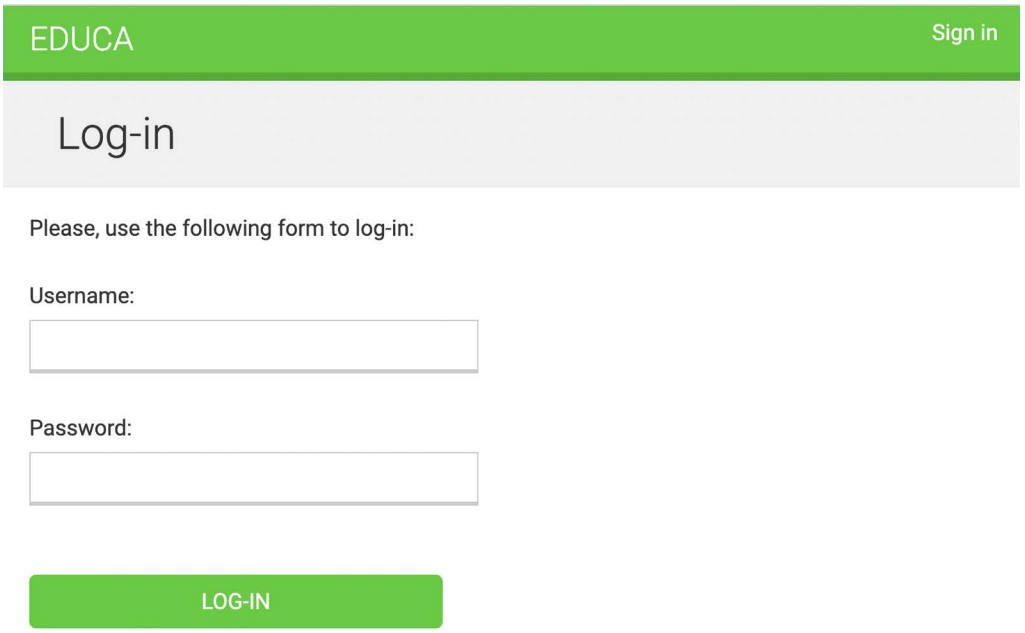


Figure 12.4: The account login page

Open `http://127.0.0.1:8000/accounts/logout/` in your browser. You should see the **Logged out** page now, as shown in *Figure 12.5*:

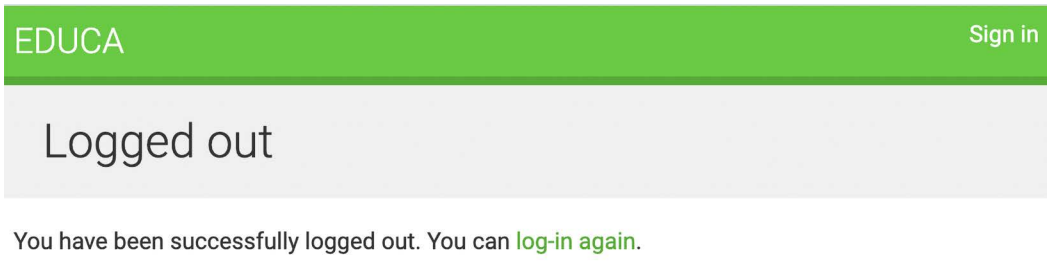


Figure 12.5: The account logged out page

You have successfully created an authentication system for the CMS.

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-4-by-example/tree/main/Chapter12`
- Using Django fixtures for testing — `https://docs.djangoproject.com/en/4.1/topics/testing/tools/#fixture-loading`
- Data migrations — `https://docs.djangoproject.com/en/4.1/topics/migrations/#data-migrations`
- Creating custom model fields – `https://docs.djangoproject.com/en/4.1/howto/custom-model-fields/`
- Static directory for the e-learning project –`https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter12/educa/courses/static`

Summary

In this chapter, you learned how to use fixtures to provide initial data for models. By using model inheritance, you created a flexible system to manage different types of content for the course modules. You also implemented a custom model field on order objects and created an authentication system for the e-learning platform.

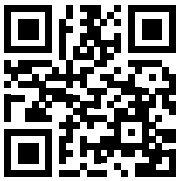
In the next chapter, you will implement the CMS functionality to manage course contents using class-based views. You will use the Django groups and permissions system to restrict access to views, and you will implement formsets to edit the content of courses. You will also create a drag-and-drop functionality to reorder course modules and their content using JavaScript and Django.

Join us on Discord

Read this book alongside other users and the author.

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 book community.

`https://packt.link/django`



13

Creating a Content Management System

In the previous chapter, you created the application models for the e-learning platform and learned how to create and apply data fixtures for models. You created a custom model field to order objects and implemented user authentication.

In this chapter, you will learn how to build the functionality for instructors to create courses and manage the contents of those courses in a versatile and efficient manner.

In this chapter, you will learn how to:

- Create a content management system using class-based views and mixins
- Build formsets and model formsets to edit course modules and module contents
- Manage groups and permissions
- Implement a drag-and-drop functionality to reorder modules and content

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter13`.

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 the requirements at once with the command `pip install -r requirements.txt`.

Creating a CMS

Now that you have created a versatile data model, you are going to build the CMS. The CMS will allow instructors to create courses and manage their content. You need to provide the following functionality:

- List the courses created by the instructor
- Create, edit, and delete courses
- Add modules to a course and reorder them

- Add different types of content to each module
- Reorder course modules and content

Let's start with the basic CRUD views.

Creating class-based views

You are going to build views to create, edit, and delete courses. You will use class-based views for this. Edit the `views.py` file of the `courses` application and add the following code:

```
from django.views.generic.list import ListView
from .models import Course

class ManageCourseListView(ListView):
    model = Course
    template_name = 'courses/manage/course/list.html'

    def get_queryset(self):
        qs = super().get_queryset()
        return qs.filter(owner=self.request.user)
```

This is the `ManageCourseListView` view. It inherits from Django's generic `ListView`. You override the `get_queryset()` method of the view to retrieve only courses created by the current user. To prevent users from editing, updating, or deleting courses they didn't create, you will also need to override the `get_queryset()` method in the create, update, and delete views. When you need to provide a specific behavior for several class-based views, it is recommended that you use *mixins*.

Using mixins for class-based views

Mixins are a special kind of multiple inheritance for a class. You can use them to provide common discrete functionality that, when added to other mixins, allows you to define the behavior of a class. There are two main situations to use mixins:

- You want to provide multiple optional features for a class
- You want to use a particular feature in several classes

Django comes with several mixins that provide additional functionality to your class-based views. You can learn more about mixins at `https://docs.djangoproject.com/en/4.1/topics/class-based-views/mixins/`.

You are going to implement common behavior for multiple views in mixin classes and use it for the course views. Edit the `views.py` file of the `courses` application and modify it as follows:

```
from django.views.generic.list import ListView
from django.views.generic.edit import CreateView, \
    UpdateView, DeleteView
from django.urls import reverse_lazy
```

```
from .models import Course

class OwnerMixin:
    def get_queryset(self):
        qs = super().get_queryset()
        return qs.filter(owner=self.request.user)

class OwnerEditMixin:
    def form_valid(self, form):
        form.instance.owner = self.request.user
        return super().form_valid(form)

class OwnerCourseMixin(OwnerMixin):
    model = Course
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin):
    template_name = 'courses/manage/course/form.html'

class ManageCourseListView(OwnerCourseMixin, ListView):
    template_name = 'courses/manage/course/list.html'

class CourseCreateView(OwnerCourseEditMixin, CreateView):
    pass

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):
    pass

class CourseDeleteView(OwnerCourseMixin, DeleteView):
    template_name = 'courses/manage/course/delete.html'
```

In this code, you create the `OwnerMixin` and `OwnerEditMixin` mixins. You will use these mixins together with the `ListView`, `CreateView`, `UpdateView`, and `DeleteView` views provided by Django. `OwnerMixin` implements the `get_queryset()` method, which is used by the views to get the base QuerySet. Your mixin will override this method to filter objects by the `owner` attribute to retrieve objects that belong to the current user (`request.user`).

`OwnerEditMixin` implements the `form_valid()` method, which is used by views that use Django's `ModelFormMixin` mixin, that is, views with forms or model forms such as `CreateView` and `UpdateView`. `form_valid()` is executed when the submitted form is valid.

The default behavior for this method is saving the instance (for model forms) and redirecting the user to `success_url`. You override this method to automatically set the current user in the `owner` attribute of the object being saved. By doing so, you set the owner for an object automatically when it is saved.

Your `OwnerMixin` class can be used for views that interact with any model that contains an `owner` attribute.

You also define an `OwnerCourseMixin` class that inherits `OwnerMixin` and provides the following attributes for child views:

- `model`: The model used for QuerySets; it is used by all views.
- `fields`: The fields of the model to build the model form of the `CreateView` and `UpdateView` views.
- `success_url`: Used by `CreateView`, `UpdateView`, and `DeleteView` to redirect the user after the form is successfully submitted or the object is deleted. You use a URL with the name `manage_course_list`, which you are going to create later.

You define an `OwnerCourseEditMixin` mixin with the following attribute:

- `template_name`: The template you will use for the `CreateView` and `UpdateView` views.

Finally, you create the following views that subclass `OwnerCourseMixin`:

- `ManageCourseListView`: Lists the courses created by the user. It inherits from `OwnerCourseMixin` and `ListView`. It defines a specific `template_name` attribute for a template to list courses.
- `CourseCreateView`: Uses a model form to create a new `Course` object. It uses the fields defined in `OwnerCourseMixin` to build a model form and also subclasses `CreateView`. It uses the template defined in `OwnerCourseEditMixin`.
- `CourseUpdateView`: Allows the editing of an existing `Course` object. It uses the fields defined in `OwnerCourseMixin` to build a model form and also subclasses `UpdateView`. It uses the template defined in `OwnerCourseEditMixin`.
- `CourseDeleteView`: Inherits from `OwnerCourseMixin` and the generic `DeleteView`. It defines a specific `template_name` attribute for a template to confirm the course deletion.

You have created the basic views to manage courses. Next, you are going to use the Django authentication groups and permissions to limit access to these views.

Working with groups and permissions

Currently, any user can access the views to manage courses. You want to restrict these views so that only instructors have permission to create and manage courses.

Django's authentication framework includes a permission system that allows you to assign permissions to users and groups. You are going to create a group for instructor users and assign permissions to create, update, and delete courses.

Run the development server using the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/auth/group/add/` in your browser to create a new `Group` object. Add the name `Instructors` and choose all permissions of the `courses` application, except those of the `Subject` model, as follows:

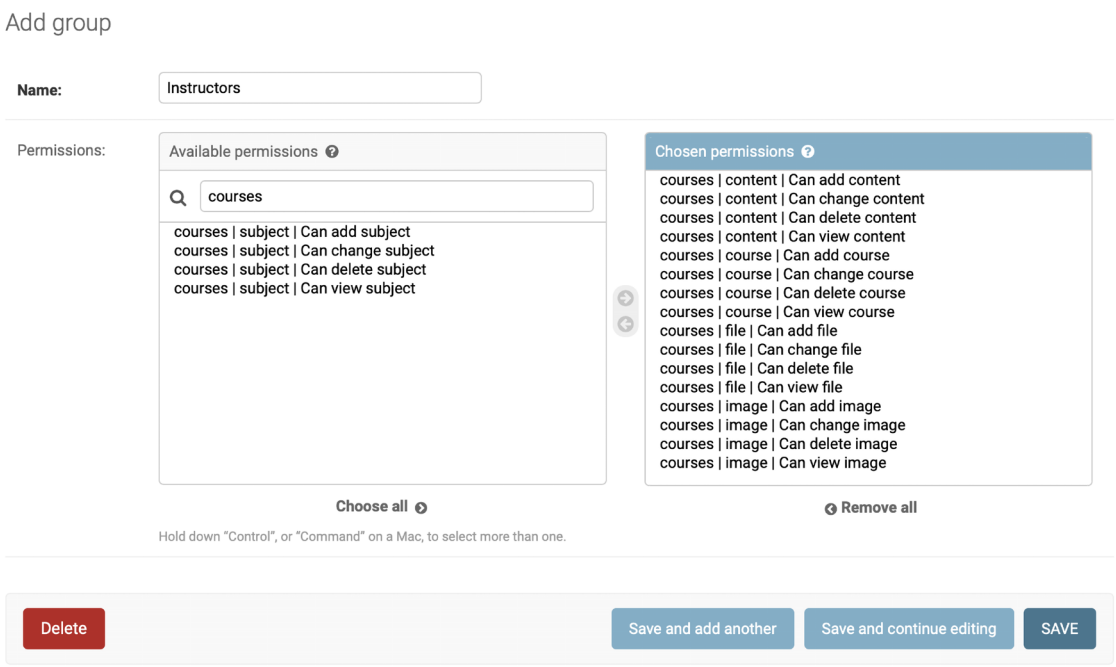


Figure 13.1: The Instructors group permissions

As you can see, there are four different permissions for each model: *can view*, *can add*, *can change*, and *can delete*. After choosing permissions for this group, click the **SAVE** button.

Django creates permissions for models automatically, but you can also create custom permissions. You will learn how to create custom permissions in *Chapter 15, Building an API*. You can read more about adding custom permissions at `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#custom-permissions`.

Open `http://127.0.0.1:8000/admin/auth/user/add/` and create a new user. Edit the user and add it to the **Instructors** group, as follows:

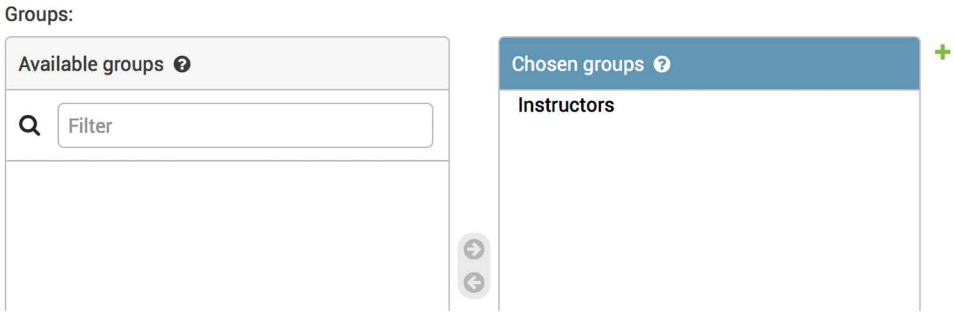


Figure 13.2: User group selection

Users inherit the permissions of the groups they belong to, but you can also add individual permissions to a single user using the administration site. Users that have `is_superuser` set to `True` have all permissions automatically.

Restricting access to class-based views

You are going to restrict access to the views so that only users with the appropriate permissions can add, change, or delete `Course` objects. You are going to use the following two mixins provided by `django.contrib.auth` to limit access to views:

- `LoginRequiredMixin`: Replicates the `login_required` decorator's functionality.
- `PermissionRequiredMixin`: Grants access to the view to users with a specific permission. Remember that superusers automatically have all permissions.

Edit the `views.py` file of the `courses` application and add the following import:

```
from django.contrib.auth.mixins import LoginRequiredMixin, \
                                       PermissionRequiredMixin
```

Make `OwnerCourseMixin` inherit `LoginRequiredMixin` and `PermissionRequiredMixin`, like this:

```
class OwnerCourseMixin(OwnerMixin,
                       LoginRequiredMixin,
                       PermissionRequiredMixin):
    model = Course
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')
```

Then, add a `permission_required` attribute to the course views, as follows:

```
class ManageCourseListView(OwnerCourseMixin, ListView):
    template_name = 'courses/manage/course/list.html'
    permission_required = 'courses.view_course'
```

```
class CourseCreateView(OwnerCourseEditMixin, CreateView):
    permission_required = 'courses.add_course'

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):
    permission_required = 'courses.change_course'

class CourseDeleteView(OwnerCourseMixin, DeleteView):
    template_name = 'courses/manage/course/delete.html'
    permission_required = 'courses.delete_course'
```

`PermissionRequiredMixin` checks that the user accessing the view has the permission specified in the `permission_required` attribute. Your views are now only accessible to users with the proper permissions.

Let's create URLs for these views. Create a new file inside the `courses` application directory and name it `urls.py`. Add the following code to it:

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

urlpatterns = [
    path('mine/',
         views.ManageCourseListView.as_view(),
         name='manage_course_list'),
    path('create/',
         views.CourseCreateView.as_view(),
         name='course_create'),
    path('<pk>/edit/',
         views.CourseUpdateView.as_view(),
         name='course_edit'),
    path('<pk>/delete/',
         views.CourseDeleteView.as_view(),
         name='course_delete'),
]
```

These are the URL patterns for the list, create, edit, and delete course views. The `pk` parameter refers to the primary key field. Remember that pk is a short for primary key. Every Django model has a field that serves as its primary key. By default, the primary key is the automatically generated `id` field. The Django generic views for single objects retrieve an object by its `pk` field. Edit the main `urls.py` file of the `educa` project and include the URL patterns of the `courses` application, as follows.