DJANGO FOR FULLSTACK WEB APPLICATIONS

ESTHER VAATI

All rights reserved
No part of this publication may be reproduced, stored or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise without written permission from the publisher. It is illegal to copy this book, post it to a website, or distribute it by any other means without permission.

ESTHER VAATI@2023

Chapter 1 : Introduction

This book follows a hand one approach to learning django by building an ecommerce web application. This book is perfect for a django developer who wants to learn how to make web applications with django quickly without the hassle of integrating with other frontend frameworks.

The first Chapter will cover the basic fundamental concepts and components of django applications. We will delve into the structure of a Django project and explore how models, views, urls, and templates interact to create dynamic web applications. You will also learn how URL routing works and how to handle HTTP requests and responses. You will also learn how Django uses Django's built-in ORM for seamless database integration.

Armed with this knowledge, you will be well-prepared to embark on more advanced topics in subsequent chapters, where we will explore authentication, form handling, Context Processor, and Payment Integration while building a fully functional ecommerce website.

For the rest of the chapters, we will build a django ecommerce application called LUX with the following features:

1. User Authentication and Registration: Users can create accounts, log in, and reset their passwords securely.
2. Product Management: A product management system where admins can add, edit and delete products
3. Shopping Cart and Checkout: A shopping cart functionality where users can add products and update quantities.
4. Payment integration with Stripe. We will also integrate a secure checkout process with Stripe payment processing gateway for online transactions.
5. Search Functionality: search feature that will allow users to search for products.

6. Wish Lists: A wishlist feature allowing users to favorite their desired items.
7. Order management. Users will be able to see their order history and their order status.
8. Email Notifications: Set up automatic email notifications for order confirmations

Prerequisites:

- You should have a basic knowledge of Python, HTML, Bootstrap and CSS and a general understanding of how web pages work.

Why Django

Django is a popular framework powering large websites such as Instagram, Pinterest, The Washington Post, and Spotify. It has also been adopted by some of the most popular online platforms, such as YouTube and Reddit, showcasing its scalability and reliability. One of the best features of django is the robustness and ease of development. Django is an excellent choice for powering high-traffic websites.

Virtual Environments

A virtual environment is a folder that allows you to install isolated Python packages for every project.For example, if you have two Django projects, each requiring a different version of Django, using a virtual environment for each project allows you to install and use the specific Django version needed for that project. This way, you can maintain compatibility and avoid conflicts between the Django versions used in different projects.

Your First Django Project

Since we have covered why we need a virtual environment, we will not install Django locally in our operating system but instead in a virtual environment. Let's create a folder to house our django application and the virtual environment.

```
mkdir django_projects
```

Next, create a virtual environment using the venv module. The venv module allows you to create and manage virtual environments by isolating packages and dependencies.

```
cd django_projects
python -m venv myenv
```

Activate the virtual environment.

```
source myenv/bin/activate
```

If you are using Windows, the command for activating your virtual environment is:

```
myenv\Scripts\activate
```

You should now see the terminal has `(myenv)` at the beginning. This shows that you are using a virtual environment. For example, mine looks like this:

```
django_projects

(myenv) vaati@vaati:~/Desktop/ django_projects
```

Install Django in the virtual environment.

```
pip install django
```

When you issue the above command, the latest version of Django will be installed, and you if all goes well, you should see something like this:

```
Collecting django
  Using cached Django-4.2.4-py3-none-any.whl (8.0 MB)
Collecting asgiref<4,>=3.6.0
  Using cached asgiref-3.7.2-py3-none-any.whl (24 kB)
Collecting sqlparse>=0.3.1
  Using cached sqlparse-0.4.4-py3-none-any.whl (41 kB)
Collecting typing-extensions>=4
  Using cached typing_extensions-4.7.1-py3-none-any.whl (33
kB)
Installing collected packages: typing-extensions, sqlparse,
asgiref, django
Successfully installed asgiref-3.7.2 django-4.2.4 sqlparse-
0.4.4 typing-extensions-4.7.1
```

If you wish to install a specific version of django, for example, Django 4.0, specify the version while installing as shown.

```
pip install django==4.0
```

To create a django project, Django comes with the `django-admin` command, which allows you to perform administrative tasks. The command looks like this.

```
django-admin <command> [options]
```

To create a new django project, issue the following command in your terminal.

```
django-admin startproject myproject
```

Where **myproject** is the name of the django project, to ensure there are no conflicts, you should not have hyphens in the name of your django project or use inbuilt function names; for example, you can't name your project django or test.

Django uses a **requirements.txt** file to list all the modules needed for the django project. A **requirements.txt** file allows the project to be reproduced on another machine. It's also essential for team collaboration. The requirements.txt file should be in the project root directory.

Issue the pip freeze command to see all the modules installed in your virtual environment.

```
pip freeze
```

Output:

```
asgiref==3.7.2
Django==4.2.4
sqlparse==0.4.4
typing_extensions==4.7.1
```

To generate a requirements.txt file, go to your project directory and issue the following command.

```
cd myproject
pip freeze > requirements.txt
```

The above command will write all the installed packages and their versions to a requirement.txt file.

Your project directory now looks like this:

```
.
├── myproject
│ ├── asgi.py
│ ├── __init__.py
│ ├── settings.py
│ ├── urls.py
│ └── wsgi.py
├── manage.py
└── requirements.txt
```

init.py -

This is an empty file that tells us that the directory myproject is a package.

asgi.py

This file contains the configurations necessary to deploy your django application with ASGI. ASGI (Asynchronous Server Gateway Interface), a successor to WSGI, handles asynchronous requests between your Djjango application and the server.

settings.py

This file contains configurations for your django application, such as database settings, middleware settings, static file settings, template settings, installed _apps, and other configurations.

urls.py

This file contains the URL dispatcher settings for your django application. The file now looks like this:

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

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

From the file above, the available paths for our django application if it were running locally are:

https://127.0.0.1:8000/admin

wsgi.py

This file contains the configurations necessary to deploy your django application with WSGI (Web Server Gateway Interface).

manage.py

manage.py is another tool provided by django that allows you to perform django tasks such as running the server, creating django apps, running tests, e.t.c

Django development server

You have just created your first Django application. To see it running locally, issue the following command.

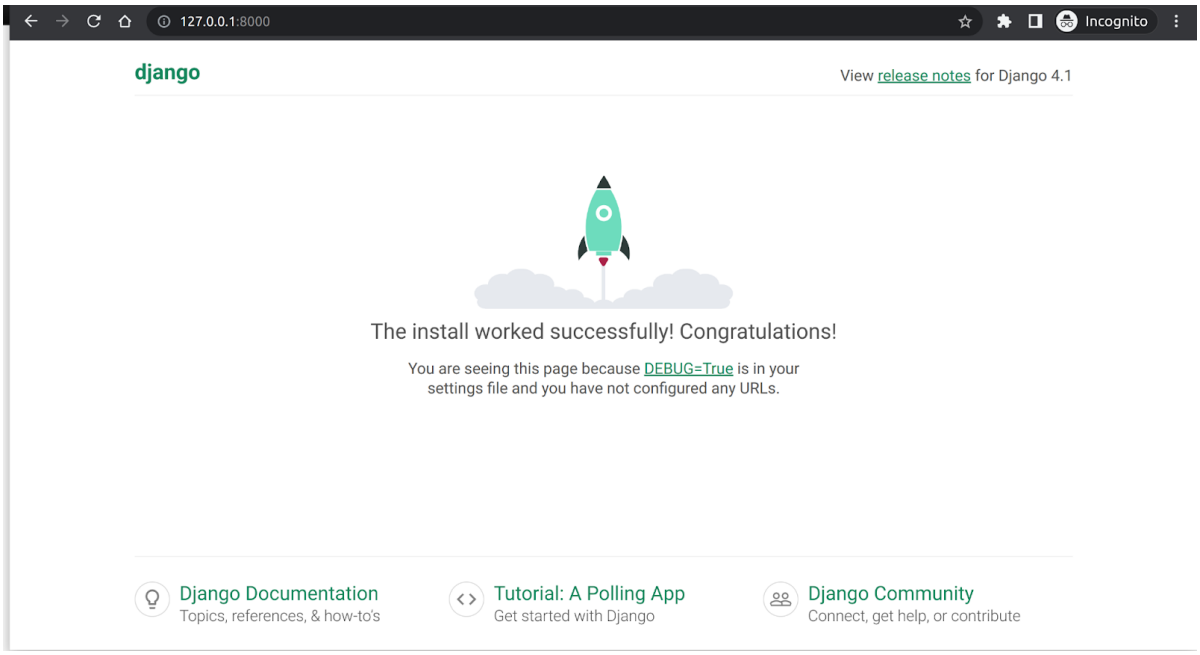
```
python manage.py runserver
```

You should see something like this on the terminal.

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not
work properly until you apply the migrations for app(s):
admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
August 02, 2023 - 07:32:51
Django version 4.1.2, using settings 'myproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Dont worry about the migrations error, which tells you you have unapplied migrations; we will cover migrations later.

The django development server is now running at http://127.0.0.1:8000/. Using your web browser, navigate to http://127.0.0.1:8000/ and see the django default page.



This shows that your django application is running as expected. This is a development server and, therefore, not ideal for use in production. The server also watches for any changes you make in your code, so there is no need to restart the server every time you make changes.

By default, the server will run at port 8000; however, if you wish to change the port, you can specify it when running it.

```
python manage.py runserver 8001
```

Django Apps

Django allows you to separate different components of your project using apps. We use the python manage.py startapp command to create a new app, as shown below.

```
python manage.py startapp <app-name>
```

Where app-name is the app's name, let's go ahead and create a products app. Go to the root directory of your django app and issue the following command.

```
python manage.py startapp products
```

Django will create a products folder and generate starter files in the directory. The directory structure should now look like this:

```
.
├── db.sqlite3
├── myproject
│ ├── asgi.py
│ ├── __init__.py
│ ├── settings.py
│ ├── urls.py
│ └── wsgi.py
├── manage.py
├── products
│ ├── admin.py
│ ├── apps.py
│ ├── __init__.py
│ ├── migrations
│ │ └── __init__.py
│ ├── models.py
│ ├── tests.py
│ └── views.py
└── requirements.txt
```

Your django project, however, does not know about the products app, so we need to add it to the list of installed_apps in the **settings.py** file. Open the **settings.py** file and add it as shown below.

```
INSTALLED_APPS = [
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'products, # add here.
]
```

Django apps are reusable and movable, so they can be transferred and integrated into another django project.

Your first django view

Django views are used to serve and pass web requests to the server. For example, when browsing a web page, the webpage is served by a view function. Django views contain all the logic needed to get requests from the user and return the appropriate response. Django views are placed in the **views.py** file.

Some views in django will fetch data from a database or an API; you can have views that return static data (data that doesn't change). For example, let's write a view that shows the current date and time. Open **products/views.py** and add the following code.

```
from django.http import JsonResponse
def products(request):
    products = [
        {"name": "django pro","description": "django pro"},
    ]
    return JsonResponse({"products": products})
```

In the code above, First, we import the `JsonResponse` class from Django's `http` module. An HTTP response class that consumes data to be serialized to JSON. Then we define a function view called products which takes a request object as a parameter; in the products function,

we then and then define a list of products.

Finally, we return an HTTP response containing the products. To serve this view to the browser, we need to map the view to a Url. Django uses url_patterns to match views to urls; these settings are found in **urls.py**. Every app will have its own **urls.py** file. However, let's use the root **urls.py** file to serve our view. Open **urls.py** and map the products view to a url.

```
from django.contrib import admin
from django.urls import path
from products.views import products

urlpatterns = [
   path('admin/', admin.site.urls),
   path('products', products),
]
```

In the code above, we import the products function from the **products/views.py** file. If you are not familiar with imports, visit the Python documentation.

Let's run the server and see it in action.

```
python manage.py runserver
```

Now navigate to http://127.0.0.1:8000/products and see the data rendered in JSON.

Templates and Urls

We have seen how to serve simple JSON data but our application will grow, and we will need to serve our data through templates. Template configurations in django are already defined in the **settings.py** file, as shown below.

```
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',
     ],
   },
 },
]
```

The `APP_DIRS': True`, means that django will look for templates in the app's directories by default. So, for example, for our products app, we need to create a templates folder. By convention, when you create a templates folder, you should also create another folder inside with the same name as the app. All the template files will then be placed in the inner products folder. Let's create our templates directory and add a products.html page.

```
products/
└── templates/
    └── products/
        └── products.html
```

Now open products.html and add the following code.

```
<!DOCTYPE html>
<html>
   <head>
      <title>Products</title>
   </head>
   <body>
      <h1>Products</h1>
      <p>Pro Django</p>
      <p>Pro Javascript</p>
   </body>
</html>
```

Let's write a view to serve the **products.html** template; update the **products/views.py** file as follows:

```
from django.shortcuts import render

def products(request):
  return render(request, 'products/products.html')
```

In the code above, we first import the render function from `django.shortcuts`. The render function is used to render a django template. We then define a products view which takes a request object as a parameter. We then use the render function to render the **products.html** template.

As we mentioned, each django app should have its **urls.py** file; create a new file, **urls.py**, in the products app folder.

```
├── admin.py
├── apps.py
├── __init__.py
├── migrations
│ └── __init__.py
├── models.py
├── templates
│ └── products
│   └── products.html
├── tests.py
├── urls.py
└── views.py
```

Map the products function to a url. Open the **products/urls.py** file and add the following code.

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


urlpatterns = [
   path('products', views.products, name = 'products'),
]
```

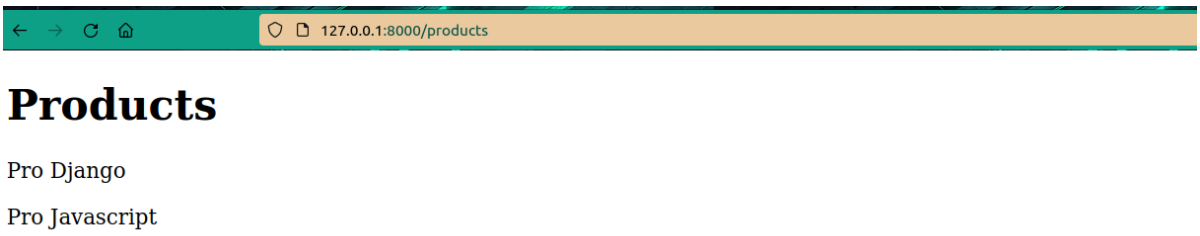
Your django project needs to know about the products urls. Open the root **urls.py** file and include the **products.urls** so that they are available for requests.

```
from django.contrib import admin
from django.urls import path,include
urlpatterns = [
   path('admin/', admin.site.urls),
   path('', include('products.urls')),
]
```

Run the development server.

```
python manage.py runserver
```

Now the server will render the content inside the products.html page, and you should see the page(http://127.0.0.1:8000/products)in your web browser.



Urls patterns

We have seen how to map views to urls; for example, let's break down the url below.

```
path('products', views.products, name = 'products'),
```

When the django development server is running, ‘home’ is the url pattern that will serve the contents of the index.html page.

- `views.products`: This view function renders the products.html page.
- `name = ‘products’` is a name assigned to the url pattern. Names in urls help uniquely identify URL patterns in django. They also make it easy to add links in templates. For example, if we needed to add a link to the products page on another page, it would look like this:

```
<a href="{% url 'products' %}">Products</a>
```

Django template engine

The django template engine is a powerful language that allows you to pass variables to templates, write conditional logic and loops, and add urls in templates.

To pass variables from the view to the template, django uses a context. A context is a dictionary containing data to be passed to a template.

Go to **products/views.py**, add a context dictionary containing a key title and the value “Pro PHP.” Pass the context to the template.

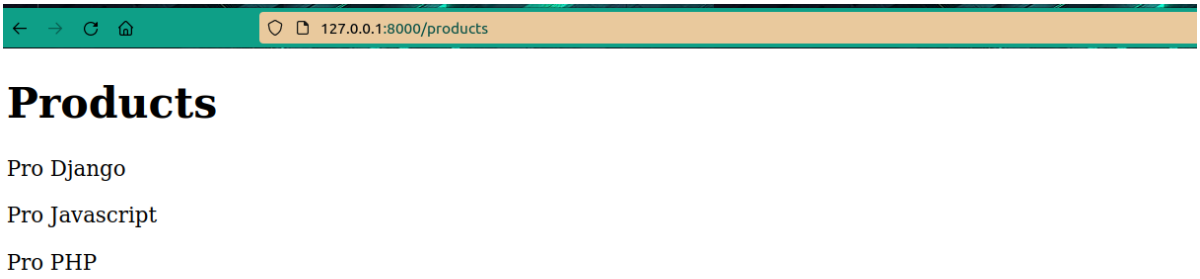
```
from django.shortcuts import render

def products(request):
  context = {'title': 'Pro PHP'}
  return render(request, 'products/products.html',context)
```

Within the **products**.html template, you can access the title variable using double curly brackets, as shown below.

```
 <!DOCTYPE html>
<html>
   <head>
      <title>Products</title>
   </head>
   <body>
      <h1>Products</h1>
      <p>Pro Django</p>
      <p>Pro Javascript</p>
      <p>{{title}}</p>
   </body>
</html>
```

Double curly brackets are template tags used for rendering variables or expressions.

Now if you refresh the page,http://127.0.0.1:8000/products you notice that the page has been updated,



You can apply django filters or any other Python expression within the tags. For example, you can apply the upper filter to make the variable uppercase. The pipe(|) character is used to apply the filter.

```
    <p>{{title|lower}}</p>
```

Loops and conditionals in templates

Data obtained from the database is usually in dictionary and list formats. To pass this data to a template, you would iterate with loops and if statements and render it to the templates. Suppose you had a list of products where each product has an id and a title. You would pass the products as a context as shown below.

```
from django.shortcuts import render

def products(request):

  products = [
      {'id':1,'title':"Pro Django"},
      {'id':2,'title':"Pro Javascript"}
       ]
  context = {'products': products}
  return render(request, 'products/products.html',context)
```

To render each product instance details, you would then write a for loop that looks like this:

```
<!DOCTYPE html>
<html>
   <head>
      <title>Products</title>
   </head>
   <body>
      <h1>Products</h1>
      {% for product in products%}
     <p>{{product.id}}</p>
     <p> {{product.title}}</p>
     {% endfor%}
   </body>
</html>
```

As you can see above, the template tags used for the for loop differ from those used to display the variables. django template engine uses the `{% %}` tags for templates and conditional statements. Just like a normal for loop, the `{% for %}` and `{% endfor %}` enclose the code to be executed .

Template Inheritance

Template inheritance is another powerful tool provided by django. Template inheritance allows you to build a base template that contains the most commonly repeated elements of your application, such as the header, navbar, footer, or any other common page elements.

The base template contains blocks that serve as placeholders for child templates. Other templates can then inherit the base template.

We will create a **base.html** template that contains a standard navbar and a block content where child templates will fill in their contents.

Earlier, we saw how to configure templates in django apps. Django also allows us to configure templates by adding `'DIRS': [BASE_DIR /'templates']` in settings.py template configurations.

```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR /'templates'], #new
        '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',
```

```
            ],
        },
    },
]
```

This configuration means that Django will also look for template files in the templates directory located inside your project's base directory (BASE_DIR).

Create the templates folder and add a base.html page.

```
.myproject
└── templates
    └── base.html
```

In the base.html page, add the contents below.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1">
    <title>My Project</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/css
/bootstrap.min.css" rel="stylesheet" integrity="sha384-
4bw+/aepP/YC94hEpVNVgiZdgIC5+VKNBQNGCHeKRQN+PtmoHDEXuppvnDJ
zQIu9" crossorigin="anonymous">
  </head>
  <body>

    <nav class="navbar bg-dark border-bottom border-body"
data-bs-theme="dark">
        <div class="container-fluid">
          <a class="navbar-brand" href="#">My Project</a>
```

```
          <button class="navbar-toggler" type="button"
data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-
controls="navbarNav" aria-expanded="false" aria-
label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
          </button>
          <div class="collapse navbar-collapse"
id="navbarNav">
            <ul class="navbar-nav">
              <li class="nav-item">
                <a class="nav-link active" aria-
current="page" href="#">Home</a>
              </li>
              <li class="nav-item">
                <a class="nav-link" href="#">Products</a>
              </li>
              <li class="nav-item">
                <a class="nav-link" href="#">SignUp</a>
              </li>

            </ul>
          </div>
        </div>
      </nav>
       {% block content %}
        {% endblock %}
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/js/b
ootstrap.bundle.min.js" integrity="sha384-
HwwvtgBNo3bZJJLYd8oVXjrBZt8cqVSpeBNS5n7C8IVInixGAoxmnlMuBnh
bgrkm" crossorigin="anonymous"></script>
  </body>
</html>
```

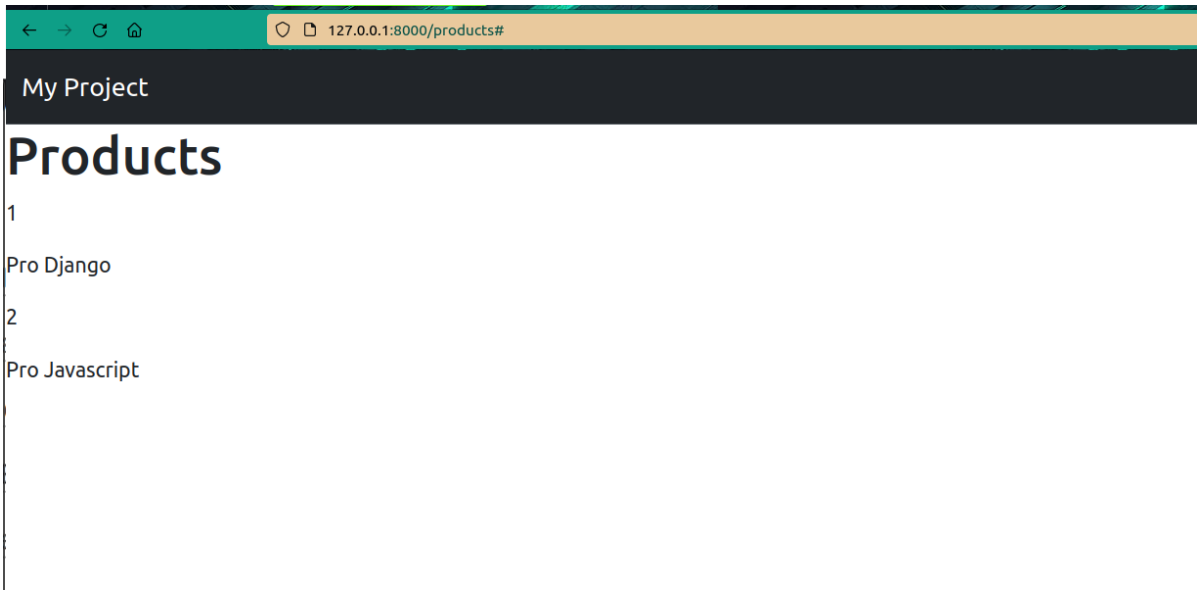
In the code above, we define a basic web page structure with a title and a nav bar using Bootstrap. As you can see after the navbar, we have `{% block content %} {% endblock %}` . Contents of child pages that extend the **base.html** page will be added inside the content block.

The **products.html** file can inherit the base template and fill the content block as shown below.

```
{% extends "base.html" %}
{% block content %}
  <h1>Products</h1>
      {% for product in products%}
          <p>{{product.id}}</p>
          <p> {{product.title}}</p>
       {% endfor%}
{% endblock %}
```

The `{% extends “base.html” %}` tag means that the above template inherits all the base template’s contents and replaces the content block with the contents of the child template.

The products page now looks like this:



Static files

Static files in django include images, CSS, and javascript files.

Static files are stored in the static folder of your django apps.Add the following files in the root directory.

```
.static
└── css
  └── main.css
```

In the settings.py file, add the following.

```
STATICFILES_DIRS = [BASE_DIR / 'static']
```

`STATICFILES_DIRS = [BASE_DIR / 'static']` means django will search for static files in the static directory. By storing the static files in the static folder, we keep your project clean and organized.

Add the load static tag in the **base.html** file to load the static files.

```
{% load static %}
<link rel="stylesheet" href="{% static 'css/main.css' %}"
/>
```

`% load static % }` is the tag used to load static files within a template. The `{% static 'css/main.css' %}` will generate the URL of the main.css file. The **main.css** file will contain the styles of how your web pages will be displayed.

Chapter 2 : Database and Models

Django provides a default SQLite database with configurations already in the settings.py file.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Django uses the built-in Object-Relational Mapping (ORM) to interact with the database. The ORM allows developers to write python code to interact with the database instead of RAW SQL queries.

SQLite database is also lightweight and easy to use.

Django uses model classes to represent tables in a database. Each model contains fields that represent columns in a database table.

Designing the database schema should be done before embarking on any development work, as it saves a lot of time when you can already visualize what the data will look like.

Let's add a Product model, open **products/models.py,** and add the following code.

```
# products/models.py

from django.db import models

class Product(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    date_created = models.DateTimeField(auto_now_add=True)
```

`CharField` and `TextField` are field types. Field types determine what kind of data is stored in the database. For example, to store a name, which is a string, Django uses the `Charfield`. A `TextField` is used to store large amounts of text in the database. To store the datetime object, Django uses a `DateField` or `DateTimeField`.

Some field types take in some arguments; for example, `Charfield` requires the max_length, which specifies the number of characters of the field. `CharField` can also take in additional arguments, such as null and blank.

```
 first_name = models.CharField(max_length= 100, null =
False, blank = False)
```

- `Null` = False means that the field will not take any null value, and the reverse is True
- `blank`=False implies that the field can't be blank when working with forms, i.e., it should not be empty

The Product model is now complete. By default, Django will give each model a primary key. A primary key is a unique number that identifies each instance of a model.

Migrations

After designing our first model, the next thing django requires us to do is to run migrations. Migrations are a way of instructing the database on how our structure will look.

Migrations in Django provide the rules of writing to the database. The two migration commands are:

```
python manage.py makemigrations
python manage.py migrate
```

The makemigrations command creates a migrations file that provides the rules by which the database tables are created. Let's issue this command in the terminal

```
python manage.py makemigrations
```

If no error occurs, you should see something like this:

```
Migrations for 'products':
  products/migrations/0001_initial.py
    - Create model Product
```

As you can see above, a file called **0001.initial.py** has been created in the migrations folder. This file contains the structure of the database table.

The `makemigrations` folder only creates a blueprint of how our Product table will look; let's issue the migrate command, which will create the database tables.

```
python manage.py migrate
```

You should see something like this:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes,
products, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices...
OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages...
OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying products.0001_initial... OK
  Applying sessions.0001_initial... OK
```

As you can see above, django applies migrations for the products app and the inbuilt apps that come with django (admin, auth, contenttypes, sessions) and creates database tables for these apps. Every time you make a change to our models, we will need to run migrations.

Database Relationships

To represent model relationships, django uses the following database relationships

- One-to-one relationship
- One-to-many relationships
- Many-to-many relationships

In our products app, we can add a category field in the product model and represent the relationship using a foreign key.

Add a Category model and the category as a Foreign key field to the product model as shown below.

```
class Category(models.Model):
    name = models.CharField(max_length=200)


class Product(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    date_created = models.DateTimeField(auto_now_add=True)
    category = models.ForeignKey(Category,
on_delete=models.PROTECT,null=True)
```

Foreign keys must have an on_delete attribute. The on_delete attribute determines what happens when the associated relationships object is deleted. The field category is set to `on_delete=models.PROTECT` prevents deletion of related items.

Run migrations for the Category model

```
python manage.py makemigrations
```

Output

```
Migrations for 'products':
  products/migrations/0001_initial.py
    - Create model Product
```

```
python manage.py migrate
```

Output

```
Migrations for 'products':
  products/migrations/0002_category_product_category.py
    - Create model Category
        - Add field category to product
```

Dunder methods

Dunder methods in Django give more information about a django model. Dunder methods like **str** give a human-readable representation of the model instance in the Django admin site.

Let's add a `__str__` method to our models. Update **models.py** to look like this:

```
class Category(models.Model):
    name = models.CharField(max_length=200)

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

```
class Product(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    date_created = models.DateTimeField(auto_now_add=True)
    category = models.ForeignKey(Category,
on_delete=models.PROTECT,null=True)

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

The Django ORM

The Django ORM provides a wrapper that allows you to make database queries. To get started making queries, we will use the Python interactive shell. Issue the following command in your terminal to start the Python interactive shell.

```
python manage.py shell
```

The python shell has django functionalities and allows us to interact with data in the database. In the shell prompt, import the models from the products app.

```
>>> from products.models import Category,Product
>>>
```

Creating objects

Let’s create a new product with the title “Pro Django”

```
>>> product1 = Product.objects.create(title = 'Pro Django')
>>> product1
<Product: Pro Django>
>>>
```

In the code above, we create an instance of a Product called product1 and saved it to the database.When you print the instance again, you get the string representation of the object (this is possible because we defined a `__str__` method in the model.

You can also query attributes like the date_created and description.

```
>>> product1.date_created
datetime.datetime(2023, 8, 2, 12, 55, 46, 991178,
tzinfo=datetime.timezone.utc)

>>> product1.description
''
```

As you can see, the description is empyt since we didnt add any value to the description field.

Updating objects

Let's add a Category 'django' and update the category field of the product instance we just added.

```
>>> category1 = Category.objects.create(name = 'django')
>>> product1.category = category1
```

If you query the product category, you can see it has been updated.

```
>>> product1.category
<Category: django>
```

Deleting objects

To delete a product or category object, issue the delete() method on the object.

```
>>> product1.delete()
(1, {'products.Product': 1})
```

If you try to delete a Category, you will get this error.

```
  raise ProtectedError(
django.db.models.deletion.ProtectedError: ("Cannot delete
some instances of model 'Category' because they are
referenced through protected foreign keys:
'Product.category'.", {<Product: Pro Django>})
>>>
```

This error occurs because set on_delete=models.PROTECT on the category field. You can change the on_delete attribute if you dont need this setting. Update the category field as follows.

```
category =
models.ForeignKey(Category,on_delete=models.SET_NULL,null=T
rue)
```

By setting `on_delete=models.SET_NULL`, the category field is set to None when the category is deleted.

After making any changes to the model, you need to run migrations.

Retrieving objects

You can also retrieve all the objects in the database using the `all()` method. Let's retrieve all product instances.

```
>>> Product.objects.all()
<QuerySet [<Product: Pro Django>]>
>>>
```

You can also get the total number of products in the database with the count method, as shown below.

```
>>> Product.objects.all().count()
1
>>>
```

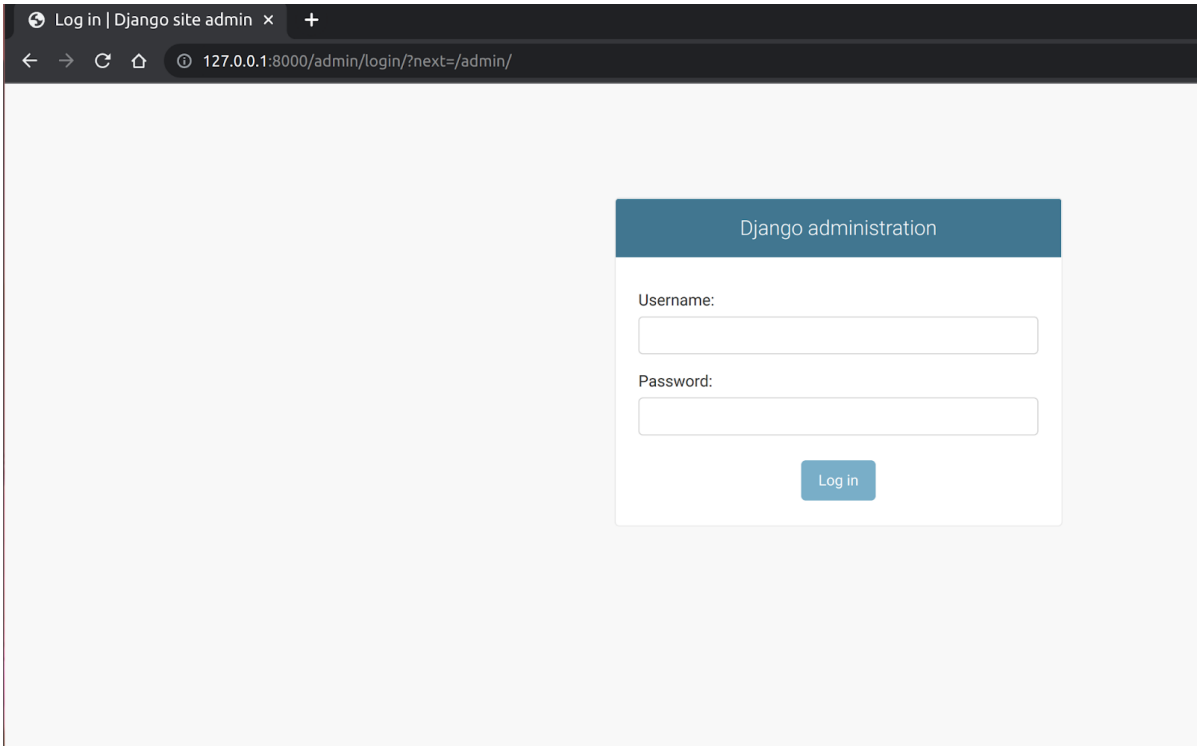
Django Admin

We have covered how to use the django ORM to add and manipulate data; we can also add data via the django admin site.

Django has an admin site that provides an interface for managing your project data. The url for the django admin site is already set in the root **urls.py** file, as shown below.

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

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

If you run the server and navigate to the admin url at http://127.0.0.1:8000/admin you should see a login screen.



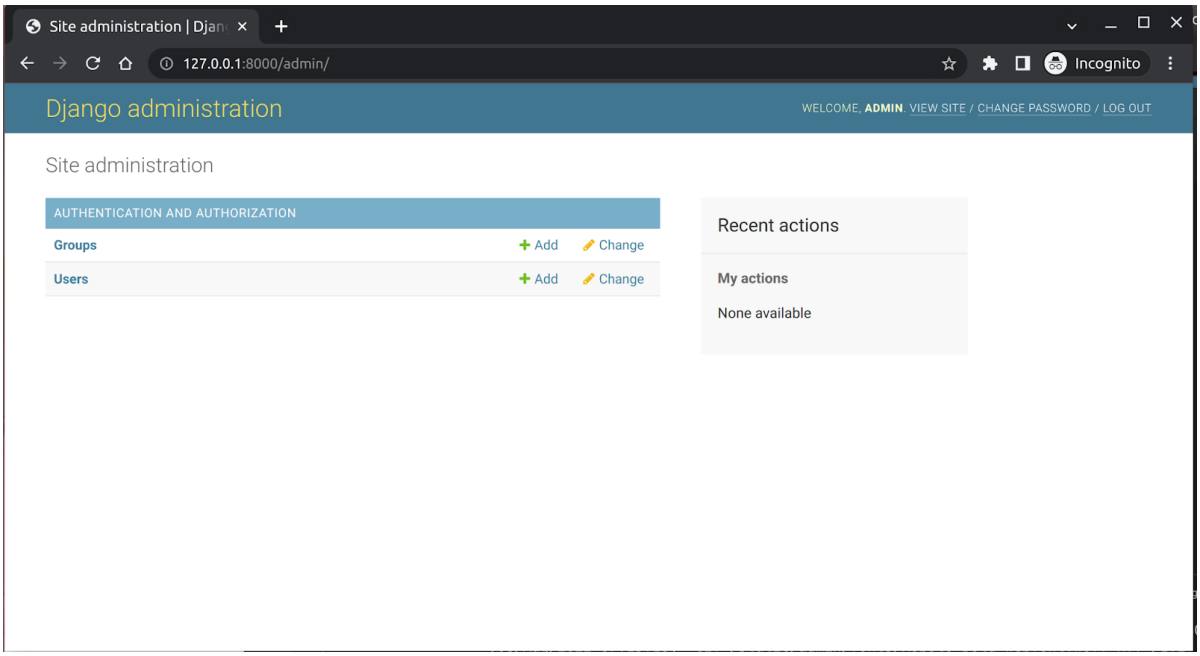
To log in to the admin site, you must be a superuser. A superuser in django is a user with admin privileges. Issue the following command in your terminal to create a super user.

```
python manage.py createsuperuser
```

You will be prompted for some details such as username, email, and password, fill them out, and when you are done, you should get a confirmation that a superuser has been created successfully

```
Username (leave blank to use admin): admin
Email address:
Password:
Password (again):
Superuser created successfully.
```

Rerun the server, navigate to http://127.0.0.1:8000/admin/, and login with the superuser credential; the site now looks like this:



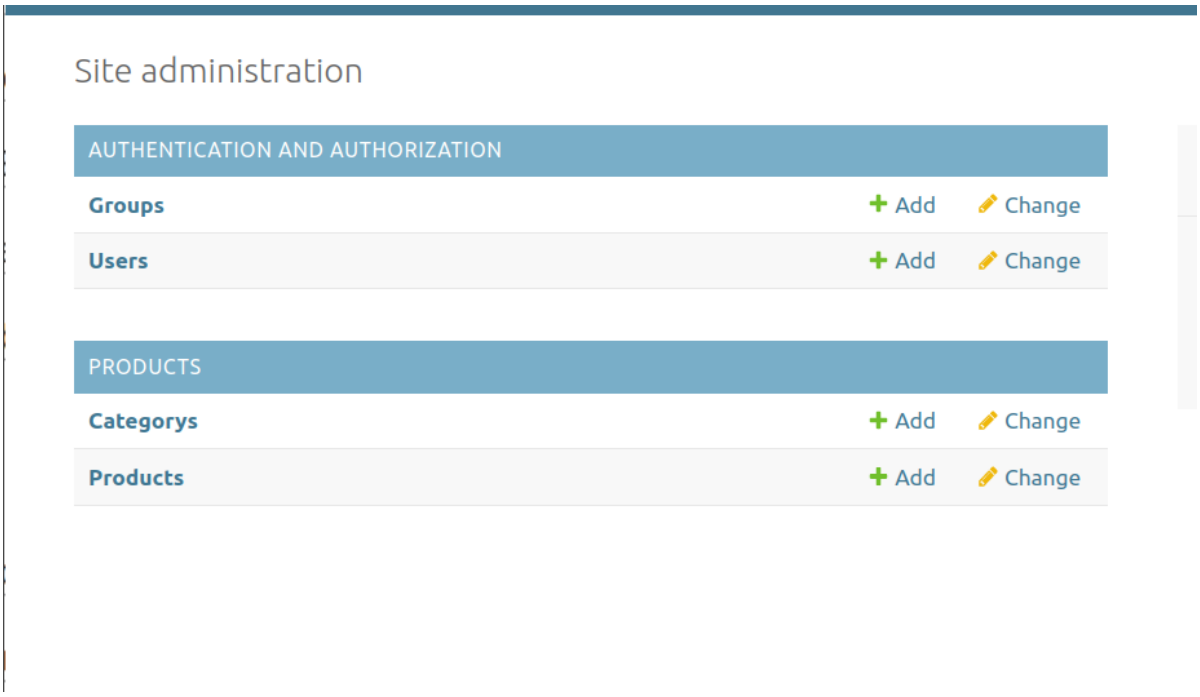
As you can see above, the Django site allows us to add and change Groups and users. We can do the same with our Category and Product models.

When we created the products app, django created for us a file called **admin.py;** we will use the **admin.py** file to register the Product and Category models.

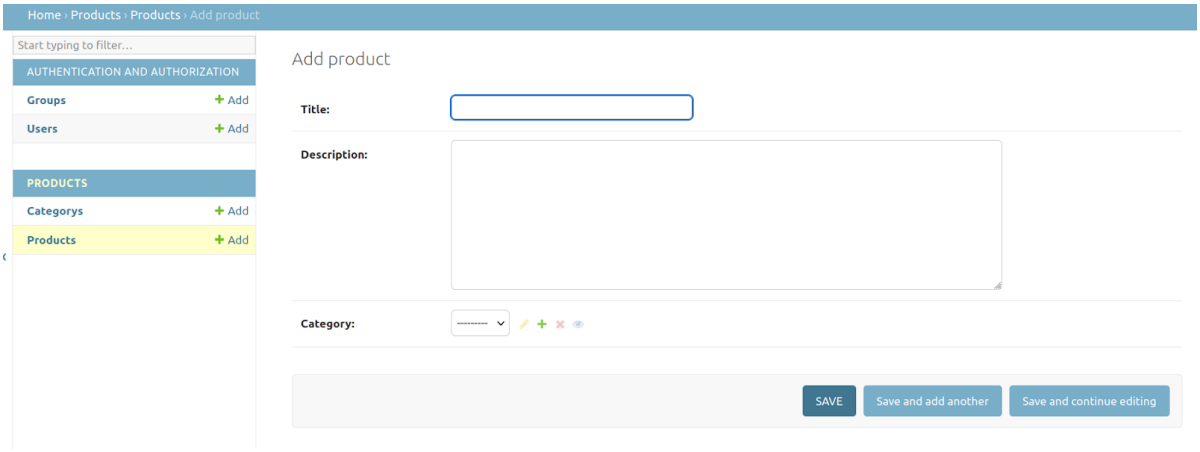
Open **the admin.py** file and add the following code.

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

admin.site.register(Product)
admin.site.register(Category)
```

Refresh the admin site.

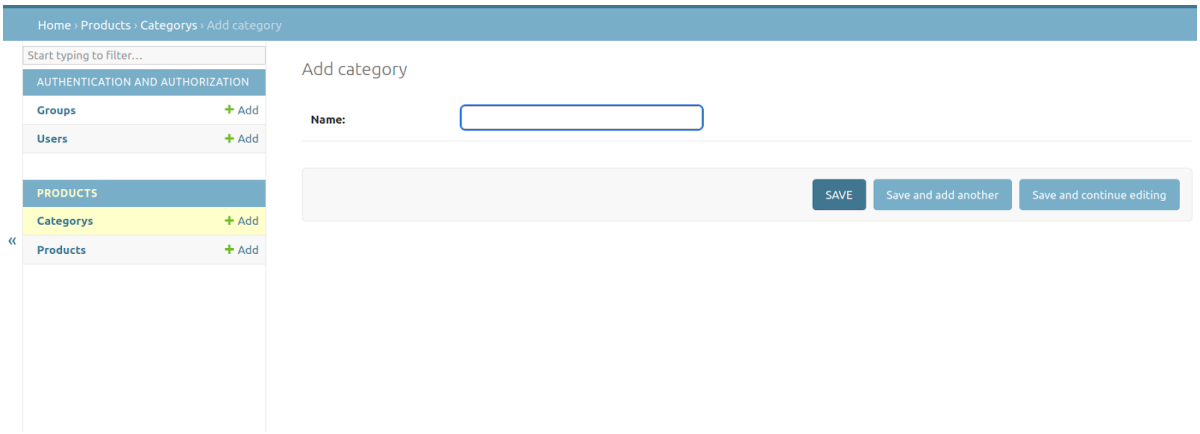


To add a Product click the Add button. You should see this form.



Fill in the data and click Save.

Add a Category.



Generic Views

We have seen how to render hardcoded data. Lets look at how to render database generated data. Django allows us to use function and classbased views. Classbased views abstract repertivie tasks such as creating, updating, reading and deleting data.

One of the most commonly used views are:

- ListView returns a page representing a list of objects
- DetailView returns page representing a single object instance.

ListView

To display a list of product objects, lets create a ProductListView which inherits from the ListView class

```
from .models import Product
from django.views.generic import ListView

class ProductListView(ListView):
    model = Product
```

By defining `model = Product`, we tell django to retrieve product objects. Once the list of objects is retrieved from the database, django will store the list of objects in the object_list variable by default.

Django will also look for a template that matches the specified model name. For example, in our case, the list of objects will be displayed in **product_list.html** unless we specify another template.

The `ListView` class also adds the object_list as context data to the template and displays the template.

Let's create the product_list.html page in the templates folder.

```
products/
└── templates/
    └── products/
        └── product_list.html
```

In the **product_list.html,** use the object_list to access the products data.

```
{% extends "base.html" %}
{% block content %}
  <h1>Products</h1>
  {% for product in object_list %}
  <ul>
      <li>{{ product.title }}</li>
      <li>{{ product.description }}</li>
      <li>{{ product.date_created }}</li>
      <li>{{ product.category }}</li>


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

Map the view to a url in the **products/urls.py** file.

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


urlpatterns = [
   path('products', views.ProductListView.as_view(), name =
'products'),
]
```

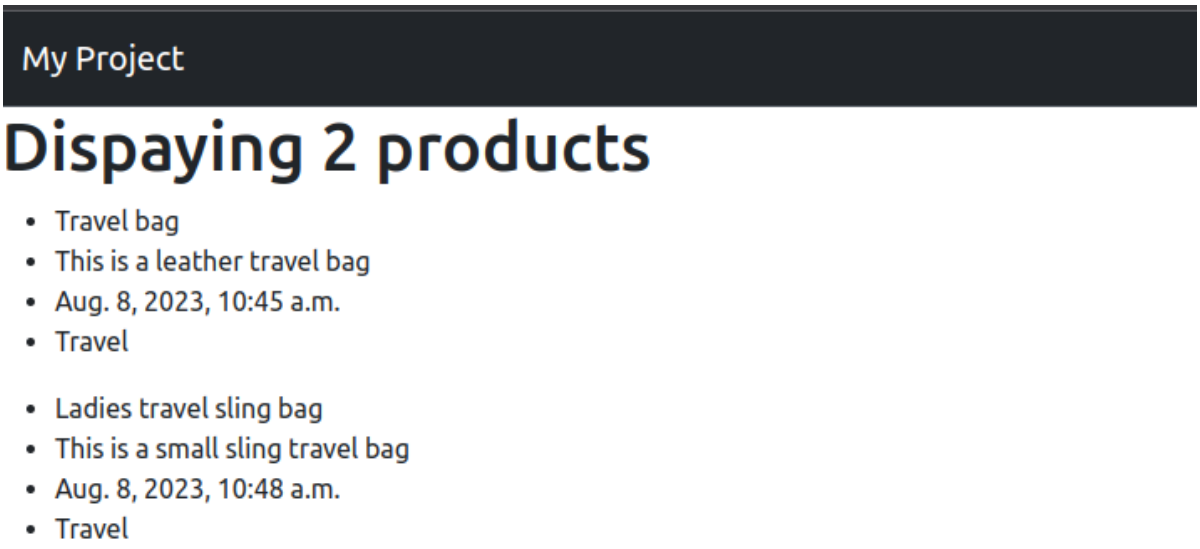
As you can see above, when defining class-based views, we use `.as_view()`, which converts the class-based view into a callable view that can be used in urls.

The `Listview` also provides methods which we can use to add different functionalities. For example the `get_context_data()` is used to add additional information to the context dictionary, for example, we can add a count of the total number of products as shown below

```
class ProductListView(ListView):
   model = Product

   def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["total_products"] =
self.model.objects.count()
        return context
```

Then in the product_list.html, access the data as follows.

```
 <h1>Dispaying {{total_products}} products</h1>
```



Other methods include:

1. setup()
2. dispatch()
3. http_method_not_allowed()
4. get_template_names()
5. get_queryset()
6. get_context_object_name()
7. get_context_data()
8. get()
9. render_to_response()

You can override these methods to cater to your specific application requirements.

DetailView

Let's create the `ProductDetailView` class. The `DetailView` works similarly to the `ListView` class. The retrieved data is stored in an object variable, and the class will use the **product_detail.html** page.

```
from django.views.generic import ListView,DetailView

class ProductDetailView(DetailView):
    model = Product
```

Create the **product_detail.html p**age in the templates folder.

```
products/
└── templates/
    └── products/
        └── product_list.html
          └── product_detail.html
```

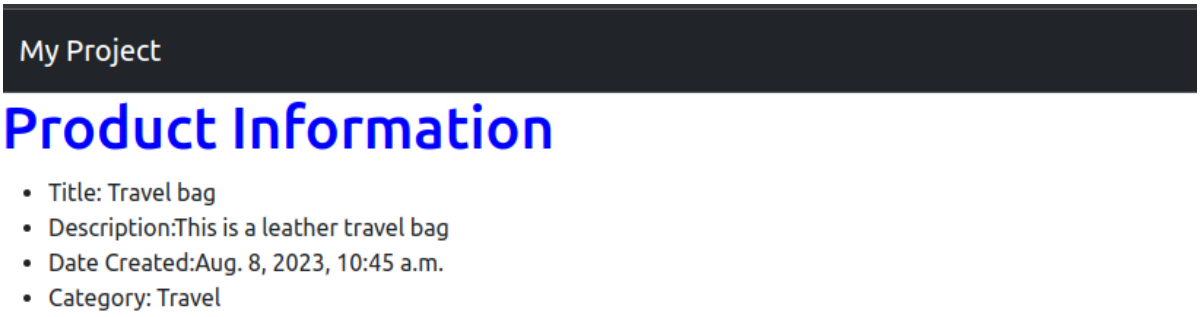
In the **product_detail.html** page, use the object variable to access the product information.

```
{% extends "base.html" %}
{% block content %}
  <h1>Product Information</h1>
  <ul>
      <li>Title: {{ object.title }}</li>
      <li>Description:{{ object.description }}</li>
      <li>Date Created:{{ object.date_created }}</li>
      <li>Category: {{ object.category }}</li>
  </ul>
{% endblock %}
```

In the **products.urls.py** file, map the view to a url.

```
urlpatterns = [
   path('products', views.ProductListView.as_view(), name =
'products'),
   path('products/<int:pk>',
views.ProductDetailView.as_view(), name = 'product-
detail'),
]
```

Since we are accessing a single product instance, we pass the primary key in the url. So now, When you navigate to a product instance, for example, with id=3 at http://127.0.0.1:8000/products/3, you should see the page below.



The `DetailView` class also comes with its own set of methods, which include:

1. setup()
2. dispatch()
3. http_method_not_allowed()
4. get_template_names()
5. get_slug_field()

6. get_queryset()
7. get_object()
8. get_context_object_name()
9. get_context_data()
10. get()
11. render_to_response()

Generic Editing Views

Generic editing views are used to create, update and delete data. These views include:

1. CreateView
2. UpdateView
3. DeleteView
4. FormView

CreateView class

The `CreateView` abstracts all the logic needed to add new model instances to the database. It displays a form for creating an object and handles saving the object to the database. In form validation, errors occur; it redisplays the form and the errors encountered. Open **products/views.py** and import `CreateView` class from `django.views.generic.edit`

```
from django.views.generic.edit import CreateView


Create a class view called ProjectCreateView.

class ProductCreateView(CreateView):
    model = Product
    fields = ["title","description",'category']
```

One major difference in the `CreateView` class is that we need to add a fields attribute which represents the fields you want displayed when you create a new object. The `CreateView` class uses a template which follows the same naming convention . So our template will be called **product_form.html.**

Create the **product_form.html in the templates folder.**

```
products/
└── templates/
    └── products/
        └── product_list.html
         └── product_detail.html
         └── product_form.html
```

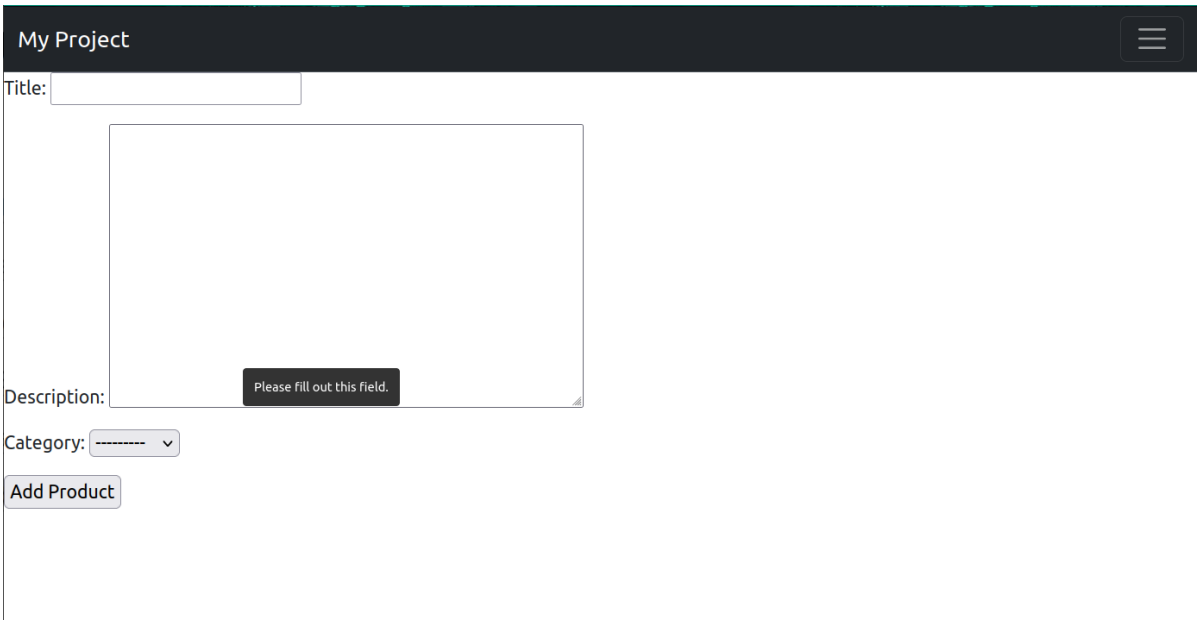
Add a form instance in the template, as shown below

```
{% extends "base.html" %}
{% block content %}

<form method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Add Product">
</form>

{% endblock%}
```

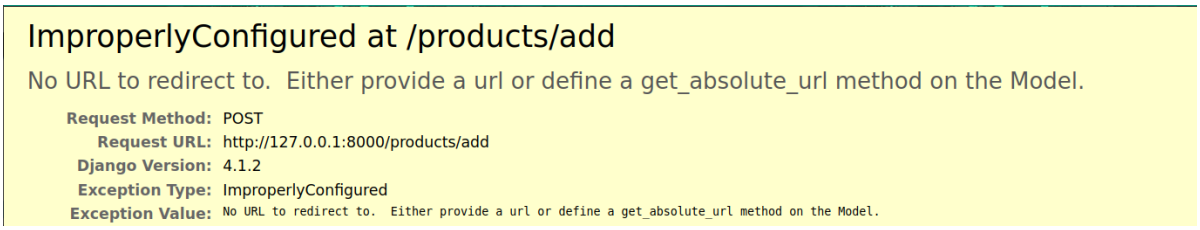
Map the view to a url in the products.urls.py file.

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


urlpatterns = [
   path('products', views.ProductListView.as_view(), name =
'products'),
   path('products/<int:pk>',
views.ProductDetailView.as_view(), name = 'product-
detail'),
   path('products/add', views.ProductCreateView.as_view(),
name = 'create-product'), #new
]
```

Navigate to http://127.0.0.1:8000/products/add , and use the form to create a new product.



After add adding a new product, you should see something like this:



This error occurs because we haven't specified a redirect url. Update the `ProductCreateView` to include the success url.

```
from django.urls import reverse_lazy
class ProductCreateView(CreateView):
    model = Product
    fields = ["title","description",'category']
    success_url = reverse_lazy('products')
```

UpdateView

The `Updateview` is a class-based view that retrieves data for an existing object and displays an editable form containing the object details. It also handles redisplaying the form with validation errors (if any) and saving changes to the object.

The form is automatically generated from the objects model unless it is specified.

Open the products/**views.py** and import the UpdateView class from django.views.generic.edit

```
from django.views.generic.edit import CreateView,
UpdateView
```

Next, add the `UpdateView` for the `Task` model.

```
class ProductUpdateView(UpdateView):
    model = Product
    fields = ["title","description"]
    template_name = 'products/product_update_form.html'
    success_url = reverse_lazy('products')
```

The `**UpdateView`** page uses a specific template with a name that ends in '_form.' For example, for our `ProductUpdateView` that updates Product objects, the default template name will be '**products/product_update_form.html**.'

We can also specify the template_name with the template_name attribute.

The success_url specifies the url where the user will be redirected after form submission.

Create the **product_update_form.html** page.

```
products/
└── templates/
    └── products/
        └── product_list.html
         └── product_detail.html
         └── product_form.html
         └── product_update_form.html
```

Add the form to the template

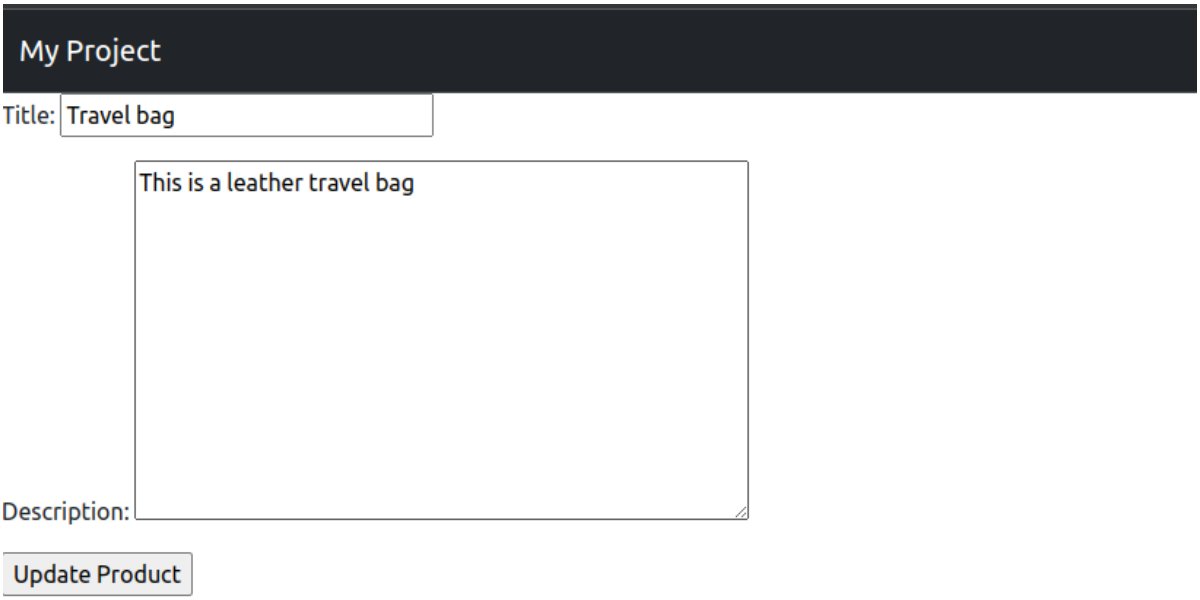
project_update_form.html

```
{% extends "base.html" %}
{% block content %}

<form method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Update Product">
</form>

{% endblock%}
```

Map the `ProductUpdateView` to aurl in the **products.urls.py** file.

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


urlpatterns = [
   path('products', views.ProductListView.as_view(), name =
'products'),
   path('products/<int:pk>',
views.ProductDetailView.as_view(), name = 'product-
detail'),
   path('products/add', views.ProductCreateView.as_view(),
name = 'create-product'),
    path('products/update/<int:pk>',
views.ProductUpdateView.as_view(), name = 'update-
product'), #new
]
```

To update a product instance for example of id =3, visit the http://127.0.0.1:8000/products/update/3 and it will it will show a prefilled form with all the details of the product instance



DeleteView

The `DeleteView` class is used to delete existing object entries. It will display a page asking the user for confirmation of deletion. The `DeleteView` doesn't take any field properties.

Import the `DeleteView` class at the top of the **products/views.py** file

```
from django.views.generic.edit import
CreateView,UpdateView,DeleteView
```

Next, add the DeleteView classes for our models

```
class ProductDeleteView(DeleteView):
    model = Product
    template_name = 'products/product_confirm_delete.html'
```

Create the **product_confirm_delete.html** page.

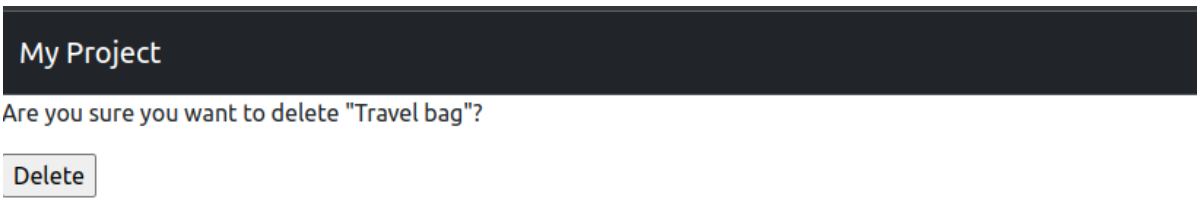
```
products/
└── templates/
    └── products/
        └── product_list.html
         └── product_detail.html
         └── product_form.html
         └── product_update_form.html
         └── product_confirm_delete.html
```

product_confirm_delete.html

```
{% extends "base.html" %}
{% block content %}

<form method="post">{% csrf_token %}
    <p>Are you sure you want to delete  "{{ object }}"?</p>
    {{ form }}
    <input type="submit" value="Delete">
</form>

{% endblock%}
```

Map the `ProductDeleteView` to a url.

```
 path('products/delete/<int:pk>',
views.ProductDeleteView.as_view(), name = 'delete-
product'),
```



Chapter 3: Build an Ecommerce Application

In this chapter and the rest of the remaining chapters of this book, we will build an ecommerce application called lux with the following features.

1. User Authentication and Registration: Users can create accounts, log in, and reset their passwords securely.
2. Product Management: A product management system where admins can add, edit and delete products
3. Shopping Cart and Checkout: A shopping cart functionality where users can add products and update quantities in their shopping cart.
4. Payment integration with Stripe. We will also integrate a secure checkout process with Stripe payment processing gateway for online transactions.
5. Search Functionality: search feature that will allow users to search for products.
6. Wish Lists: A wishlist feature allowing users to favorite their desired items.
7. Order management. Users will be able to see their order history and their order status.

Getting Started

We will start by creating a directory to house our project. Create a new directory called django_ecommerce

```
mkdir django_ecommerce
```

Create a virtual environment.

```
cd django_ecommerce
python -m venv myenv
```

Activate the virtual environment and install Django.

```
source myenv/bin/activate
pip install django
```

Create a Django project called lux. Issue the following command in your terminal.

```
django-admin startproject lux
```

Custom User

Even though django comes with a default User model, it also recommends creating a custom user model at the beginning of your project since it can take a lot of work to make customizations when your project is in its advanced stage. Let's start by creating a Custom user model.

Create a new django app called users.

```
python manage.py startapp users
```

Add the users' app to the list of installed apps in the settings.py file.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'users',
]
```

To create a Custom user, you can use either the AbstractUser or the AbstractBaseUser. The AbstactUser is more manageable, and it looks like this:

In users/models.py, add the code below

```
from django.contrib.auth.models import AbstractUser
class User(AbstractUser):
    pass
```

You might wonder “why we cannot just use the default User model `(django.contrib.auth.models.User)`; the default User model does not give customizations, and it is not extensible , this means that in the future if your project need change, you cannot customize it.

By subclassing `AbstactUse`r, you can add any other fields in the future to meet additional requirements of your django application. For example, if, in the future, you wish to add a phone field, you can add it like this:

```
class CustomUser(AbstractUser):
    phone = models.CharField(max_length=10, blank=True)
     # Add additional fields and methods
```

The AbstractUser class inherits all the fields and methods from the default User model.

Add the custom user model to the **settings.py** file so that django knows we are using a Custom user model.

```
AUTH_USER_MODEL = 'users.User'
```

Where users is the name of the app and User is the User model

Run Migrations

```
python manage.py makemigrations
```

You should see this output

```
Migrations for 'users':
  users/migrations/0001_initial.py
    - Create model User
```

Run the migrate command.

```
python manage.py migrate
```

You should see the following output.

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes,
sessions, users
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
```

```
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages...
OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying users.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices...
OK
  Applying sessions.0001_initial... OK
```

Create a superuser

```
python manage.py createsuperuser
Username: admin
Email address:
Password:
Password (again):
Superuser created successfully.
```

Product Models

Inside your lux project, create a django application called store.

```
python manage.py startapp store
```

Add the store app to the list of installed_apps in the settings.py file.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'users',
    'store',
]
```

This fictional store lux assumes that the Product will be added from the admin side, so we will create a Product model.

Open **store/models.py** file and add the code below.

```
# store.models.py
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    price = models.DecimalField(
            max_digits=10, decimal_places=2)
    image = models.ImageField(upload_to='products/')
    created_at = models.DateTimeField(auto_now_add=True)

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

The Product model with the following fields:

- **title**: represents the title of the product.
- **description**: represents more detailed information about the product.

- **price**: represents the price of the product expressed in decimals. In django, the `DecimalField` is commonly used to represent monetary values. The Decimal field should have the max_digits and decimal_place attributes. So, for example, the value stored in the price fields would look like 2.00, 2.50, 2.99, 10.30, etc
- **image** represents the product's image. The `upload_to` attribute indicates where the product images will be uploaded to.
- **created_at** shows when the product was added. `auto_now_add =True` sets the value to the date and time the product is added to the database

The `ImageField` requires us to install the Pillow library, so let's do that.

```
python -m pip install Pillow
```

The Pillow library is an image manipulation used to manipulate images in django and Python applications.

Run Migrations

```
python manage.py makemigrations
```

When you run the makemigrations command, you should see something like this,

```
Migrations for 'store':
  store/migrations/0001_initial.py
    - Create model Product
```

```
python manage.py migrate
```

The migrate command creates actual database tables. You should see this output.

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes,
sessions, store, users
Running migrations:
  Applying store.0001_initial... OK
```

Register Models to Django admin.

To add data from the django admin site, we need to register our models in the admin site. In **store/admin.py**, add the code below

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

admin.site.register(Product)
```

Static and Media Files Configurations

Before we add any data , lets configure static files and media files. By default, Django does not serve static files automatically. These settings have to be configured. In the settings.py file, add the following configurations.

```
STATIC_URL = 'static/'
STATICFILES_DIRS = [BASE_DIR / 'static']

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

- **MEDIA_URL**: this setting specifies the base URL to serve media files.

- **MEDIA_ROOT**: specifies the exact filesystem location where the image will be stored. For example, our application specified the image upload attribute, as shown below.

```
image = models.ImageField(upload_to='products/')
```

Now all the products will be stored at /media/products

When an admin uploads an image book.jpg for a product, the image url will be http://127.0.0.1:8000/media/products/book.jpg

To allow django to serve static files , you also need to add this url to your root urls.py file.

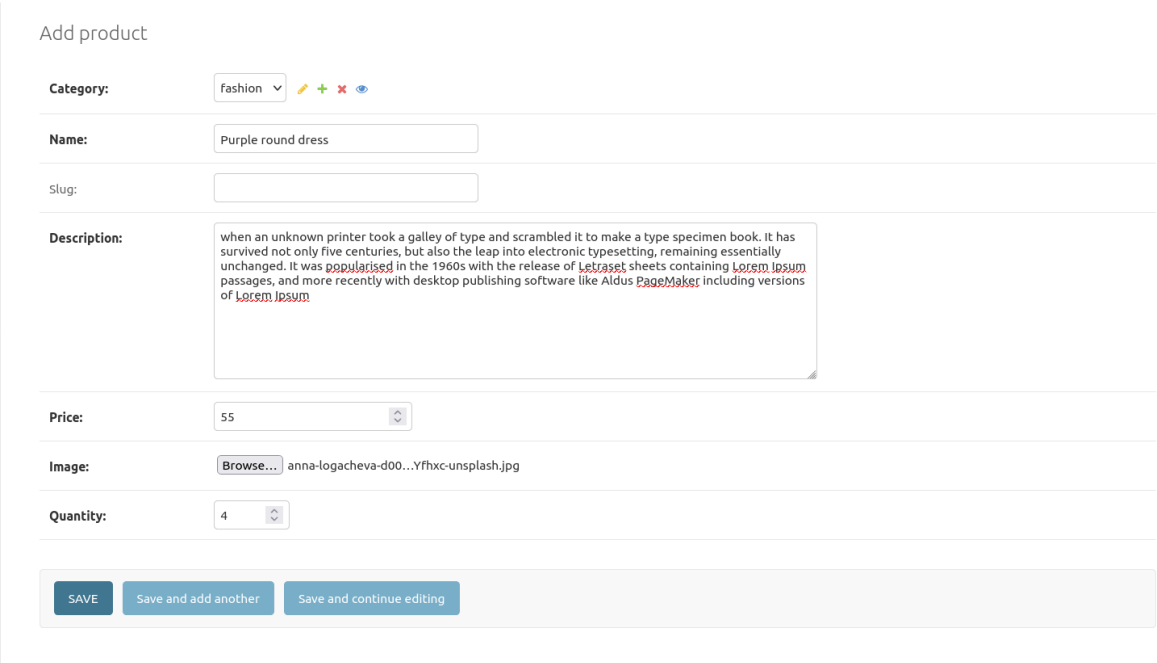
```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls)

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

Now run the development server and add products from the django admin site.

```
python manage.py runserver.
```

Navigate to localhost:8000/admin, and log in with the superuser credentials.

Add a new Product



ProductListView and ProductDetail

We need to create the following views:

- `ProductListView` -a view that fetches all the products from the database
- `productDetail` - a view that displays a single product details

Open **store/views.py** add the following code.

```
from django.shortcuts import render
from .models import Product
from django.views.generic import ListView,DetailView

class ProductListView(ListView):
    """Display all products"""

    model = Product
    template_name = 'store/product-list.html'
    context_object_name = 'products'
```

```
class productDetailView(DetailView):
    """Display product detail """
    model = Product
    template_name ='store/product_detail.html'
    context_object_name = 'product'
```

`ListView` is a generic view that allows us to display a list of items. In our case, we are using it to display products. Behind the scenes, the `ListView` class will retrieve the data for the `Product` model from the database and pass the data to a context dictionary called products.

`DetailView` is also a generic view that is used to display the details of a single model object.

By default, django will look for templates in all app directories; create the following files in the store app.

```
└── templates
      └── store
          ├── product_list.html
          └── product_detail.html
```

In the **product_list.html** file, add the following code,

```
<!-- product_list.html -->
{% extends 'base.html' %}
{% block content %}
<h1>{{ category.name }}</h1>
<div class="row">
{% for product in products %}
<div class="col-sm-4">
   <div class="card">
      <div class="card-body">
```

```
            <a href="#">
               <h5 class="card-title">{{ product.name }}</h5>
               <img src="{{ product.image.url }}" alt="{{
product.name }}" class="w-100 rounded "/>
            </a>
            <p class="card-text">${{product.price}}</p>
         </div>
      </div>
</div>
{% empty %}
<li>No products found.</li>
{% endfor %}
{% endblock %}
```

Here we loop over the `product` context and display each instance in a bootstrap card.

In your **product_detail.html** page, add the code below.

```
<!-- product_detail.html -->
{% extends 'base.html' %}
{% load static %}
{% block content %}
<div class="container mt-4">
   <div class="row">
      <div class="col-md-4">
         <img src="{{ product.image.url }}" alt="Winter
Landscape" class="w-100 rounded "/>
      </div>
      <div class="col-md-8">
         <h3>{{ product.name }}</h3>
         <strong>
            <p>$ {{ product.price }}</p>
         </strong>
         <button type="button" class="btn btn-warning btn-
lg add-to-cart-btn">Add to Cart</button>
      </div>
```

```
    </div>
</div>
<div class="container mt-4">
    <div class="row">
        <p>{{ product.description }}</p>
    </div>
</div>
{% endblock %}
```

In both templates, we are extending from the **base.html,** which contains the common elements such as the navbar, header, and footer

In the root directory, create a templates folder and add the **base.html** file.

```
.lux
└── templates
    └── base.html
```

We need to tell our project to look for templates in the root templates folder by updating `‘DIRS’: [BASE_DIR /‘templates’]` in the settings.py file.

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

In the **base.html file**, add the code below, which contains a navbar, bootsrap CDN files, and a block content section to fill content from templates that inherit this template.

```
<!doctype html>
<html lang="en">
   <head>
      <meta charset="utf-8">
      <meta name="viewport" content="width=device-width,
initial-scale=1">
      <title>LUX</title>
      <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.15.4/css/all.min.css">
      <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css
/bootstrap.min.css" rel="stylesheet" integrity="sha384-
9ndCyUaIbzAi2FUVXJi0CjmCapSmO7SnpJef0486qhLnuZ2cdeRhO02iuK6
FUUVM" crossorigin="anonymous">
   </head>
   <body style="background-color: rgb(240, 237, 237);">
      <nav class="navbar navbar-expand-lg "
style="background-color: #e8127d;">
         <div class="container-fluid">
            <a class="navbar-brand" href="#">LUX</a>
            <button class="navbar-toggler" type="button"
data-bs-toggle="collapse" data-bs-
target="#navbarSupportedContent" aria-
controls="navbarSupportedContent" aria-expanded="false"
aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
            </button>
            <div class="collapse navbar-collapse"
id="navbarSupportedContent">
               <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                  <li class="nav-item">
                     <a class="nav-link active" aria-
current="page" href="/">Shop</a>
```

```
                </li>
              </ul>
             <form action = " " method="post" class="d-
flex" role="search">
                 <input class="form-control me-2"
type="search"
                        name="search"
placeholder="Search" aria-label="Search">
                <button class="btn btn-outline-success"
type="submit">Search
                </button>
            </form>

              <ul class="navbar-nav">
                <a class="nav-link" href="">
                <i class="fa fa-shopping-cart"></i>
                <span class="cart-count"></span>
                </a>
                {% if user.is_authenticated %}
                <li class="nav-item">
                   <a class="nav-link" href="#">Hello
{{user}}</a>
                </li>
                <li class="nav-item">
                   <a class="nav-link" href="">Logout</a>
                </li>
                {% else %}
                <li class="nav-item">
                   <a class="nav-link"
href="#">SignUp</a>
                </li>
                {% endif %}
              </ul>
           </div>
        </div>
     </nav>
     <div class="container mt-4">
        <div class="row justify-content-center">
           <div class="col-md-10">
              {% block content %}
```

```
                {% endblock %}
            </div>
         </div>
      </div>
      <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js" integrity="sha384-geWF76RCwLtnZ8qwWowPQNguL3RmwHVBC9FhGdlKrxdiJJigb/j/68SIy3Te4Bkz" crossorigin="anonymous"></script>
      <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.8/dist/umd/popper.min.js" integrity="sha384-I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc2pM8ODewa9r" crossorigin="anonymous"></script>
      <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.min.js" integrity="sha384-fbbOQedDUMZZ5KreZpsbe1LCZPVmfTnH7ois6mU1QK+m14rQ1l2bGBq41eYeM/fS" crossorigin="anonymous"></script>
   </body>
</html>
```

Create a file **urls.py** in the store app.

In your **store/urls.py** files, map the views(ProductListView and ProductDetailView) to urls.

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


urlpatterns = [

    path('',views.ProductListView.as_view(), name='product-
list' ),
    path('<int:pk>',views.productDetailView.as_view(),
name='product-detail' )

    ]
```

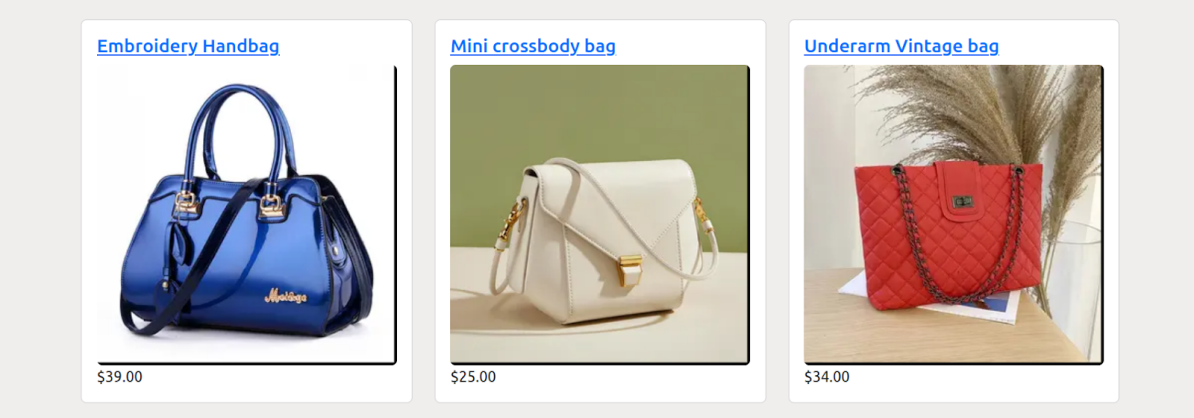
In the root urls.py file, include the URL's for the store app.

```
from django.urls import path,include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',include('store.urls'))

 ]
```

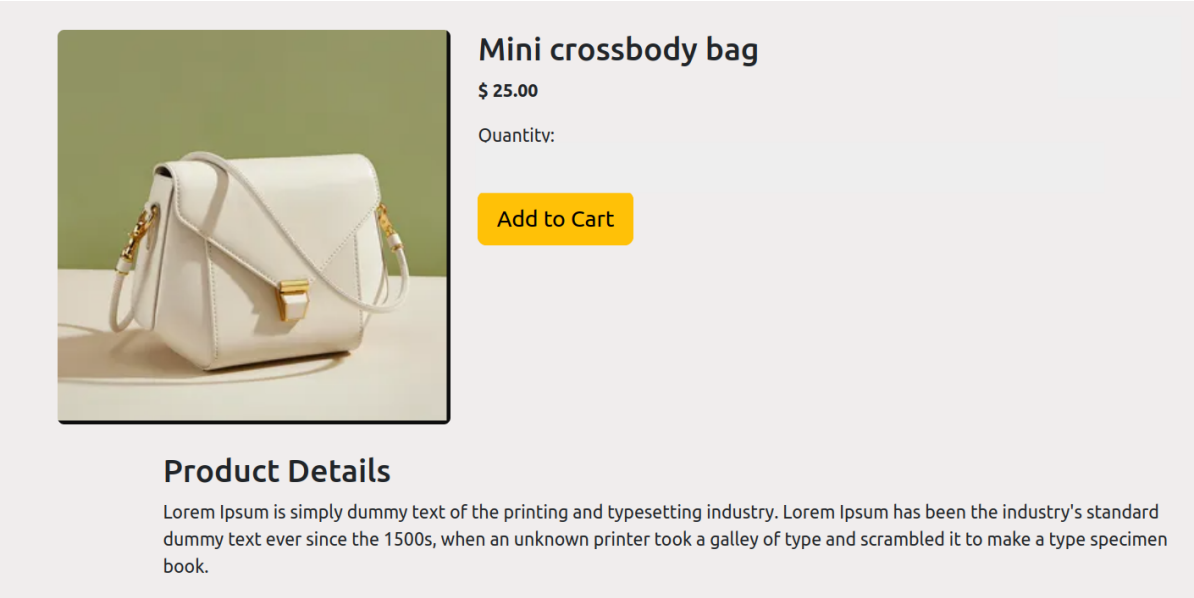
Run the development server

```
python manage.py runserver
```

Now if you navigate to http://127.0.0.1:8000/ products page looks like this;



The product detail page now looks like this:



Add To Cart

As you can see above, we have an add to cart button; items placed in the cart will be temporarily stored in the database for a while, and when a customer completes an order, the items in the cart will be deleted.

Let’s create a Cart model. A cart model will contain the following information.

- customer - the id of the currently logged-in user
- product - the id of a product
- quantity- the number of products selected by the user
- total_cost(): this is a method that will calculate the total cost of the cart items for a particular user

Open **store/models.py file** and add the information about the cart.

```
from django.conf import settings

class Cart(models.Model):
    customer = models.ForeignKey(settings.AUTH_USER_MODEL,
 on_delete=models.CASCADE)
    product = models.ForeignKey(Product, on_delete=
models.CASCADE)
    quantity = models.PositiveBigIntegerField(default=0)


    @property
    def total_cost(self):
        total =0
        cart_total = self.product.price * self.quantity
        total +=cart_total
        return cart_total

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

- `customer` is a foreign key linking to the custom user model.
- `product`: a foreign key linking to the product model
- `quantity`: a positive integer field for the number of products added to the cart. The default is set to 1.

- `total_cost` is a method that calculates the total cost of the items in the cart for a particular customer. The `@property` attribute lets us call the total_cost just like a normal field.

Perform migrations. Issue the make migrations command

```
python manage.py makemigrations
```

Outtput

```
Migrations for 'store':
  store/migrations/0002_cart.py
    - Create model Cart
```

Then run the migrate command.

```
python manage.py migrate
```

Output

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes,
sessions, store, users
Running migrations:
  Applying store.0002_cart... OK
```

Add to Cart Form

The add to cart form will allow users to add products to the cart model. In the store app, create a **forms.py** file and add a **CartForm**, which looks like this:

```
from django import forms
from .models import Cart

class CartForm(forms.ModelForm):
    class Meta:
        model = Cart
        fields = ['quantity']
        widgets = {
            'quantity': forms.NumberInput(attrs={'class':
'form-control'})
        }
```

Django comes with the `ModelForm`, a helper class that allows you to build forms from models. By specifying the Cart model in the Meta class, django will automatically create a form matching the model instance fields; in our case, the `CartForm` is built from the Cart model. We only want to display the quantity field. We can also add additional attributes, such as the widgets attribute, which will display a number input field with a form control class to the quantity field that accepts only integer values.

Add to Cart View

In your **store/views.py** file , update the **ProductDetail** class a shown below.

```
from .forms import CartForm

class productDetail(DetailView):
    """Display product detail and a cart Form"""
    model = Product
    template_name ='product_detail.html'


    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        form = CartForm()
        context['form'] = form
        return context
```

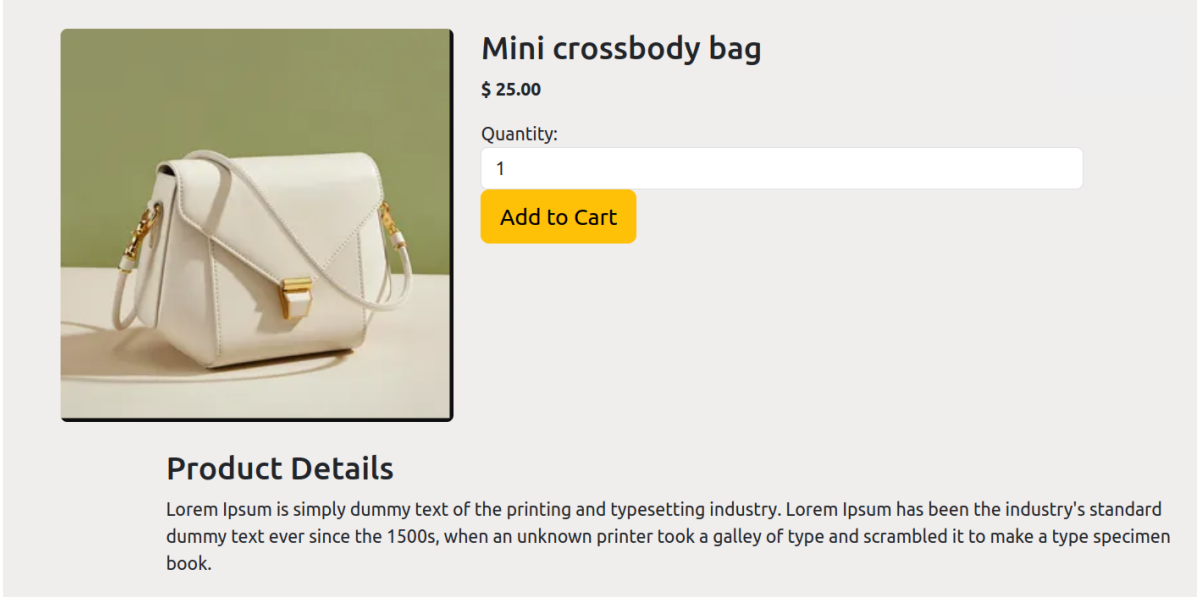
In addition to displaying product details, we added the `get_context_data()` method, which adds additional data to the template context. Here we add an instance of the `CartForm` to the context dictionary.

In the **product_detail.html** page, wrap the **Add To Cart** button with a form as shown below.

```
<form method="POST">
   {% csrf_token%}
    {{form.as_p}}
    <button type="submit" class="btn btn-warning btn-lg
add-to-cart-btn">Add  to Cart</button>
 </form>
```

Whenever a user clicks the "**Add to Cart**" button, we want to pass the **product_id** in the request data.

The product detail now has a quantity input field where the user can select the total quantities they want to order for a product.



For the **Add to Cart button** to actually add a product to the cart model, we need to implement that functionality. In **store/views.py**, add the `add_cart` function, which looks like this:

```
from .models import Product,Cart
def cart_add(request, product_id):
    """Add a product to the cart"""
    if request.method == 'POST':
        form = CartForm(request.POST)
        if form.is_valid():
            cart = form.save(commit=False)
            cart.customer = request.user
            cart.product_id = product_id
            cart.save()

    return redirect('cart-list')
```

In the **add_cart** function above, product_id is obtained from the request data.

- If the request. method is POST; we create a new `cartForm` instance using the data from the POST request.

- If the form is valid, we create a new cart object. By setting `commit=False`, the cart object is not yet saved to the database.
- `cart.customer = request.user` sets the customer to the currently authenticated user. We also set `cart.product_id = product_id` to specify the product being added to the cart. Then, we call `cart.save()`, which adds the new cart instance to the database. Finally, we redirect the user to the cart list page.

Map the cart_add view to a url in the store/urls.py file.

```
urlpatterns = [

    path('',views.ProductListView.as_view(), name='product-
list' ),
    path('<int:pk>',views.productDetail.as_view(),
name='product-detail' ),
    path('cart_add/<int:product_id>',views.cart_add,
name='cart-add' ), #new

    ]
```

In the **product_detail.html** page, add the form action to point to the cart_add url.

```
<form action="{% url 'cart-add' product.id %}"
method="POST">
 {% csrf_token%}
 {{form.as_p}}
 <button type="submit" class="btn btn-warning btn-lg add-
to-cart-btn">Add to Cart</button>
</form>
```

Cart List

Before testing the functionality, let's create the `cart_list` function to see how the products will look when added to the cart. The cart list will show the total items in the cart and the total cost. In **store/views.py**, add the cart_list function.

```
def cart_list(request):
    """retrieve cart items"""
    total_cost =0
    cartitems = Cart.objects.filter(customer=request.user)
    for item in cartitems:
        subtotal = item.total_cost
        total_cost+=subtotal


    context =
{'cart_items':cartitems,'total_cost':total_cost }
    return render(request, 'store/cart_list.html', context)
```

- We initialize a variable `total_cost` to 0, for tracking the cumulative cost of all items in the cart.
- `cartitems = Cart.objects.filter(customer=request.user)`: retrieves all cart items belonging to the currently authenticated user
- `for item in cartitems:` : Iterates through each cart item in `cartitems` queryset.
- `subtotal = item.total_cost`: for each cart item, we fetch the value of the `total_cost` method (total_cost is defined in the Cart model class)
- `total_cost += subtotal:` Adds the value of subtotal of each cart instance to the `total_cost` variable. `total_cost` is the cumulative total of all items in the cart.

- Finally, we add the **cartitems** and **total_cost** to the context dictionary

In your templates folder, create the **cart_list.html page**

```
└── templates
    └── store
        ├── product_list.html
        ├── product_detail.html
        └── cart_list.html
```

In the **cart_list.html** page, render the context data.

```
{% extends 'base.html'%}
{% block content%}
{% if cart_items %}

<div class="container mt-4">
    <div class="row">
       <h4 class="text-center">Cart Summary</h4>
    </div>
</div>

<div class="container mt-4">
    <div class="table-responsive">
   <table class="table table-bordered">
      <tbody>
         <tr>
            {% for item in cart_items %}

            <td><img src="{{ item.product.image.url }}"
alt="{{ item.product.name }}" style="max-height: 100px;">
                  </td>
            <td>{{ item.product.name }}</td>
            <td>
```

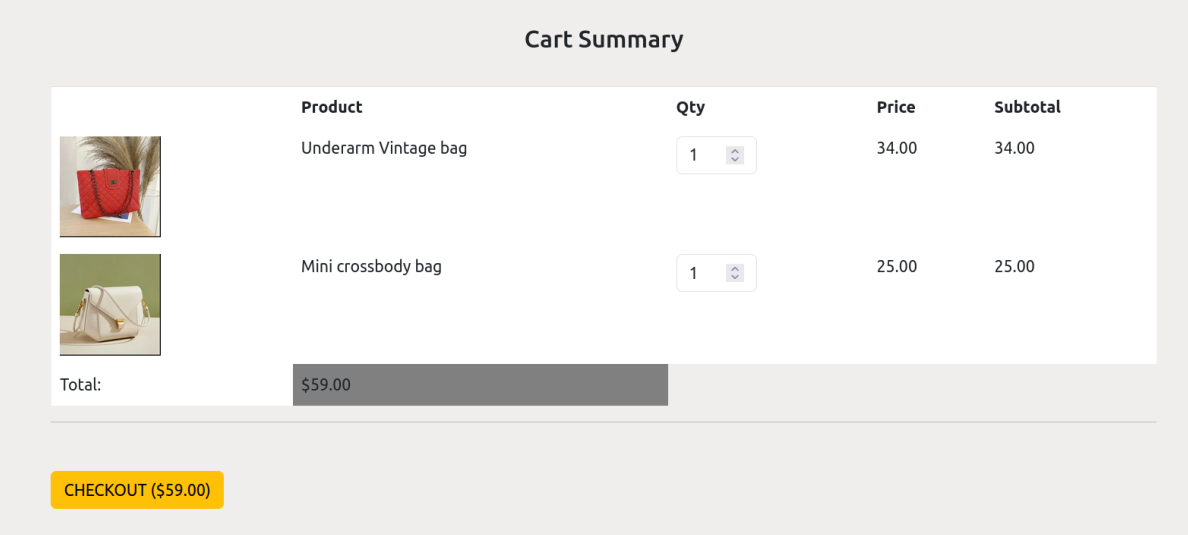
```
                <input type="number" name="quantity"
value="{{ item.quantity }}"
                       class="form-control mr-2"
style="width: 80px;" min="1"
                 >
            </td>
            <td>{{ item.product.price }}</td>
            <td>{{ item.total_cost }}</td>
         </tr>
         {% endfor %}
         <tr>
            <td>Total:</td>
            <td style=" background-color: grey;">${{
total_cost }}</td>
         </tr>
      </tbody>
   </table>
   <hr>
   <div class ="row mt-5">
      <div class="col-sm-6">
         <a href="" class="btn btn-warning">CHECKOUT
(${{total_cost}})</a>
      </div>
   </div>
</div>
</div>
{% else %}
<div class="container mt-5 text-center">
   <div class="row">
      <div class="col-md-6 offset-md-3">
         <div class="card">
            <div class="card-body">
               <p class="card-text">Your Cart is empty</p>
               <a href="/" class="btn btn-primary">Continue
Shopping</a>
            </div>
         </div>
      </div>
   </div>
</div>
```

```
{% endif %}
{% endblock%}
```

Now let's map the `cart_list` and the `add_cart` views to urls and test the functionality.

```
urlpatterns = [

    path('',views.ProductListView.as_view(), name='product-list' ),
    path('<int:pk>',views.productDetail.as_view(), name='product-detail' ),
    path('cart_add/<int:product_id>',views.cart_add, name='cart-add' ), #new
    path('cart_list',views.cart_list, name='cart-list' ), #new

    ]
```

Now if you click on the add to cart button, the item will be added to the cart, and you will be redirected to the cart_list page.



In the product_detail page, we want to add a condition,

If the featured product is already in the cart, we dont want to show the **"Add to cart"** button, but a **"Check Cart"** button. Update the `ProductDetailView` class , as follows.

```
class productDetail(DetailView):
    """Display product detail and a cart Form"""
    model = Product
    template_name ='store/product_detail.html'


    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        form = CartForm()

        #get the product instance.
        product = self.get_object()
        cart_item = Cart.objects.filter(
            customer = self.request.user,
            product=product
            )
        if cart_item.exists():
            cart =cart_item.first()
        else:
            cart = None

        context['form'] = form
        context['cart'] = cart
        return context
```

Open the **product_detail.html** page, and update it to include the condition.

```
 <div class="col-md-8">
         <h3>{{ product.name }}</h3>
         <strong>
            <p>$ {{ product.price }}</p>
         </strong>
         {% if cart %}
         <div class ="row mt-5">
            <div class="col-sm-6">
                <p>This product is already in the cart</p>
                    <a href="{% url 'cart-list'%}"
class="btn btn-warning btn-lg">Check Cart</a>
            </div>
         </div>
         {% else %}
         <form action="{% url 'cart-add' product.id %}"
method="POST">
            {% csrf_token%}
            {{form.as_p}}
            <button type="submit" class="btn btn-warning
btn-lg ">Add to Cart</button>
           </form>

            {% endif %}
       </div>
```

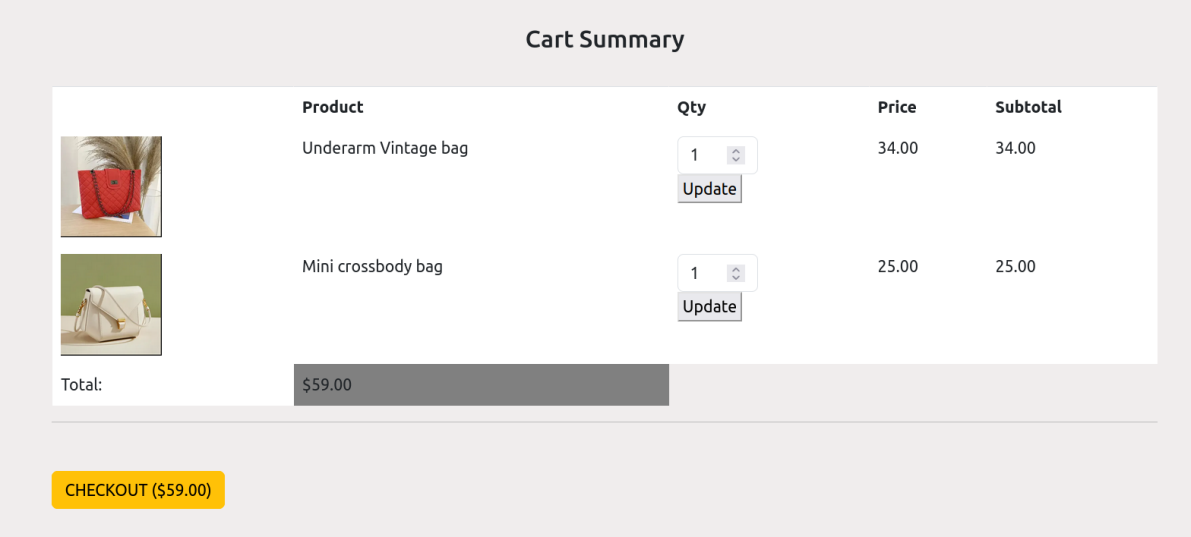
If the value of the cart is True, we display a message that informs the user that the product is already in the cart; if the cart value is NONE, we display an Add To Cart button.

Update Cart Items

On the quantity row in the **cart_list.html** page, we have an input field that shows the current quantity and allows the user to add more or reduce quantities of the same product. Update it to include a form like this:

```
<td>
   <form action = " " method="post">
      {% csrf_token %}
      <input type="text" name="id" value="{{
item.product.id }}" class="form-control d-none">
       <input type="number" name="quantity" value="{{
item.quantity }}" class="form-control mr-2" style="width:
80px;" min="1">
      <button type="submit">Update</button>
      </form>
</td>
```

The page now features an update button for each cart item.



In the `update_cart_item` view, we want to get the product's id in the request and use it to update the cart quantity. In your **store/views.py** file, add the `update_cart_item` function

```
def update_cart_item(request):
    """Update cart quantities"""
    if request.method =='POST':
        quantity = request.POST.get('quantity')
        product_id = request.POST.get('id')
        cart = Cart.objects.get(
            customer =request.user,
            product_id=product_id
            )
        cart.quantity=quantity
        cart.save()

    return redirect('cart-list')
```

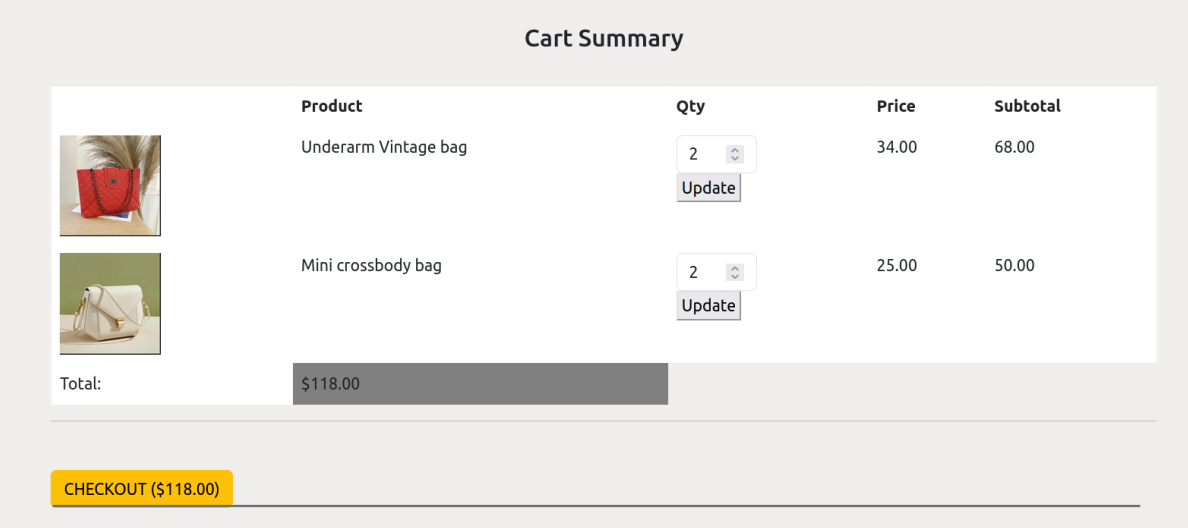
Map the view to a url

```
urlpatterns = [

    path('',views.ProductListView.as_view(), name='product-
list' ),
    path('<int:pk>',views.productDetail.as_view(),
name='product-detail' ),
    # cart urls/
    path('cart_add/<int:product_id>',views.cart_add,
name='cart-add' ),
    path('cart_list',views.cart_list, name='cart-list' ),
    path('update_item',views.update_cart_item,
name='update-item' ), #new


    ]
```

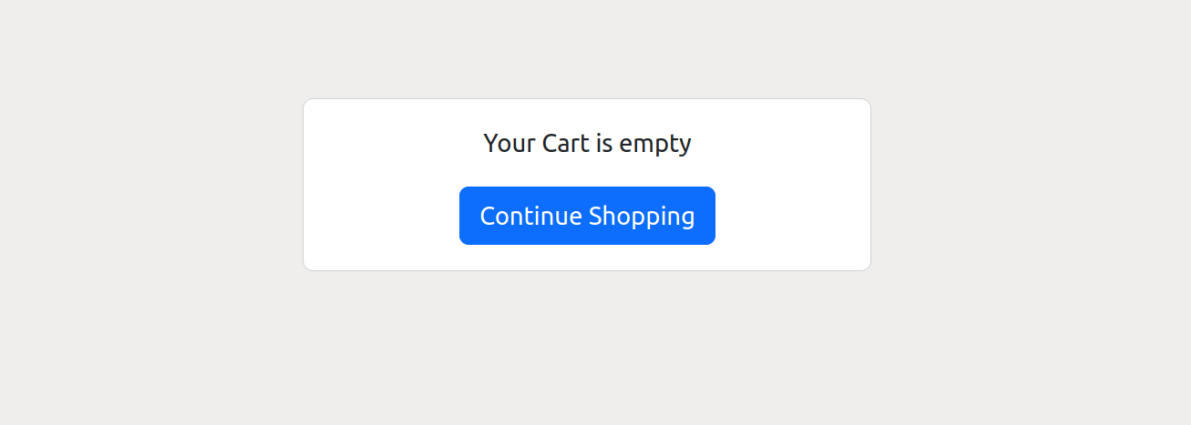
In the **cart_list.html** page, update the form action to point to the `update-item` url.

```
<td>
   <form action = "{% url 'update-item'%}" method="post">
      {% csrf_token %}
      <input type="text" name="id"
             value="{{ item.product.id }}"
             class="form-control  d-none">
       <input type="number" name="quantity"
              value="{{ item.quantity }}"
              class="form-control mr-2"
              style="width: 80px;" min="1">
      <button type="submit">Update</button>
      </form>
</td>
```

Now if you change the quantity and click the update button, the subtotal will change, and the total cost at checkout will change to reflect the new items added to the cart.



If the cart is empty, the user will see this page.



Remove Item from Cart

This functionality will delete a cart instance from the cart model. In your **store/ views.py** file, add the `remove_cart_item` function, which looks like this:

```
def remove_cart_item (request):
    """Removes cart item from cart list"""
    if request.method =='POST':
        product_id = request.POST.get('id')
        cart = Cart.objects.get(
            customer =request.user,
            product_id=product_id
             )
        cart.delete()

    return redirect('cart-list')
```

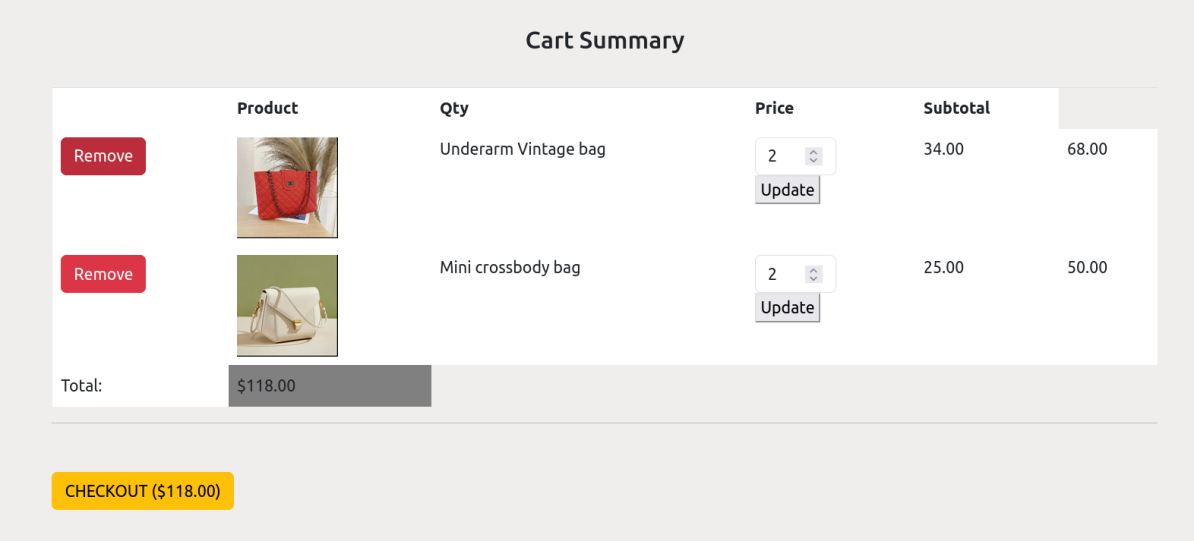
Map the view to a url in store/urls.py file

```
urlpatterns = [

    path('',views.ProductListView.as_view(), name='product-
list' ),
    path('<int:pk>',views.productDetail.as_view(),
name='product-detail' ),
    # cart urls/
    path('cart_add/<int:product_id>',views.cart_add,
name='cart-add' ),
    path('cart_list',views.cart_list, name='cart-list' ),
    path('update_item',views.update_cart_item,
name='update-item' ),
    path('remove_item',views.remove_cart_item,
name='remove-item' ), #new

    ]
```

In the **cart_list .html** page, add this code just after the `{% for item in cart_items %}`

```
<td>
   <form action = "{% url 'remove-item'%}" method="post">
      {% csrf_token %}
      <input type="text" name="id" value="{{
item.product.id }}" class="form-control d-none">
      <button class="btn btn-danger"
 type="submit">Remove</button>
   </form>
</td>
```

The cart list page now features a working button that will delete the item from the cart model when clicked.



Search For Products.

In the **base.html** page, we have a search form which contains a search placeholder.

```
<form action = " " method="post" class="d-flex"
role="search">
  <input class="form-control me-2" type="search"
name="search" placeholder="Search" aria-label="Search">
  <button class="btn btn-outline-success"
type="submit">Search</button>
</form>
```

This form will allow users to search for products. We need to implement the search functionality. Open **store/views.py** and add the code below

```
def search(request):
    if request.method =='POST':
        query = request.POST.get('search','')
        products = Product.objects.filter(
                  name__icontains =query
                       )

    return render(request, 'store/product_search.html',
{'products': products})
```

When a user adds a search query in the search form and clicks the search button, we obtain the query and filter the Product model based on the search query.

The queryset `Product.objects.filter(name__icontains =query)` will search the database where the name field contains the user query. `__icontains` is a django lookup type that performs a case-insensitive search. For example, the query django or DJANGO will return the same results.

Finally we add the resulting products queryset to a context dictionary.

Create the **product_search.html** page in the templates folder.

```
store/
└── templates/
    └── store/
        ├── cart_list.html
        ├── product_detail.html
        ├── product_list.html
        ├── product_search.html
```

In the **product_search.html** , loop through the products return from the query and display the product details, If the search results are empty, we will display the message “No products matching your Query “

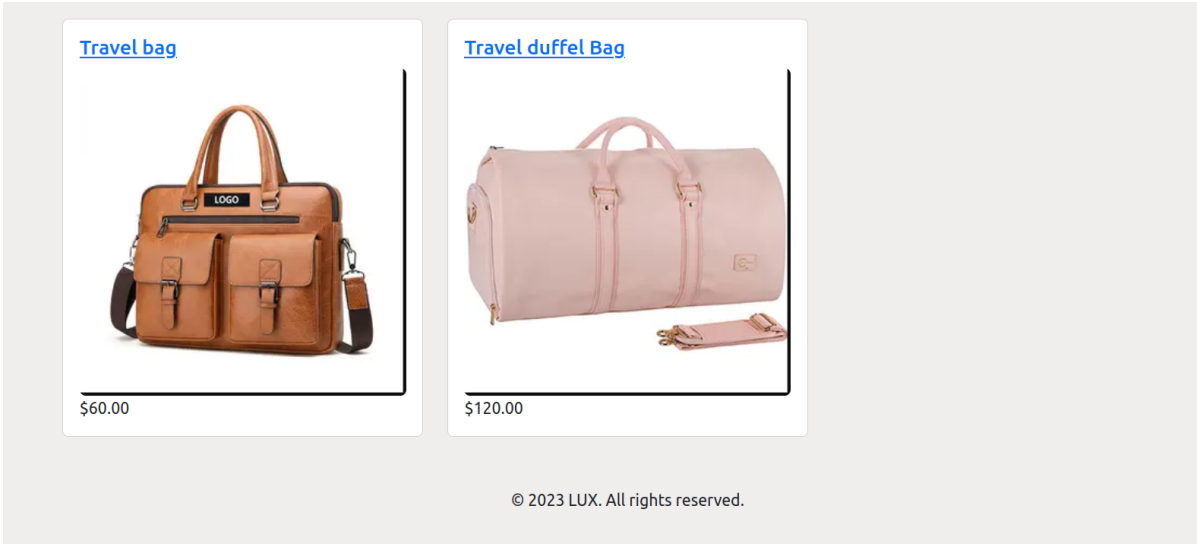
```
<!-- product_search.html -->
{% extends 'base.html' %}
{% block content %}
<div class="row">
{% for product in products %}
<div class="col-sm-4">
   <div class="card">
      <div class="card-body">
         <a href="{% url 'product-detail' product.id%}">
            <h5 class="card-title">{{ product.name }}</h5>
            <img src="{{ product.image.url }}" alt="{{
product.name }}" class="w-100 rounded "/>
         </a>
         <p class="card-text">${{product.price}}</p>
      </div>
   </div>
</div>
{% empty %}

<div class="container mt-5 text-center">
   <div class="row">
      <div class="col-md-6 offset-md-3">
         <div class="card">
            <div class="card-body">
               <p class="card-text">No products matching
your Query </p>
               <a href="/" class="btn btn-primary">Continue
Shopping</a>
            </div>
         </div>
      </div>
   </div>
</div>
```

```
{% endfor %}
{% endblock %}
```

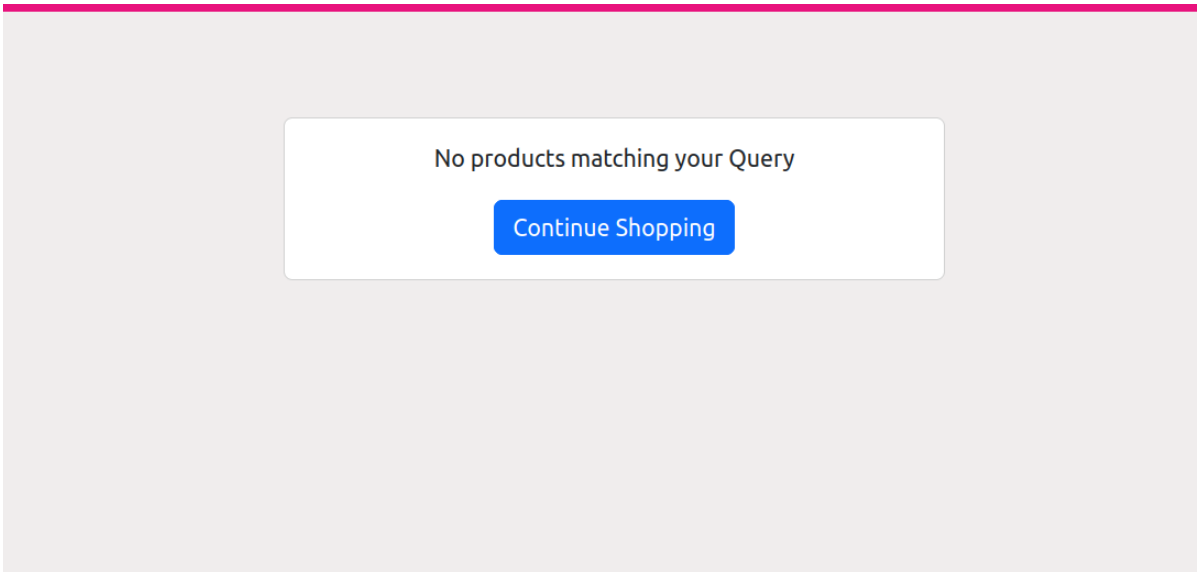
Define the url for the search view in the store/urls.py file.

```
# store/urls.py

    path('search',views.search, name='search' ), #new
```

So For example, if our search term is travel, we will get the following results,



If no results are found, the user will see this section of the page.



Go back to the **base.html** page and add define the action attribute to point to the search url and add a csrf token . The form now looks like this:

```
<form action = "{% url 'search' %}" method="post" class="d-
flex" role="search">
  {% csrf_token%}
  <input class="form-control me-2" type="search"
name="search" placeholder="Search" aria-label="Search">
  <button class="btn btn-outline-success"
type="submit">Search</button>
</form>
```

Advanced Search with Q Objects

We have performed a basic search by searching the name field. To make results more conclusive and relevant. We might employ an advanced search mechanism where we search in multiple fields. Django provides advanced search techniques like the Q objects, which give you more control over your queryset.

Q objects allow you to combine 2 filter conditions. For example, if we wanted our queryset to look for search terms in the name and description fields.

```
from django.db.models import Q
products = Product.objects.filter(Q(name__icontains =query)
| Q(description__icontains=query))
```

The pipe symbol (|) means OR, so if our search term is not found in the name field, it will also look in the description field; if a match is found, the products matching the query will be added to the result;

Update the seach function to include the advanced query.

```
from django.db.models import Q

def search(request):
    if request.method =='POST':
        query = request.POST.get('search','')
        products = Product.objects.filter(Q(name__icontains
=query) | Q(description__icontains=query))
    return render(request, 'store/product_search.html',
{'products': products})
```

Context Processors

According to Django documentation,

context_processors is a list of dotted Python paths to callables that are used to populate the context when a template is rendered with a request. These callables take a request object as their argument and return a dict of items to be merged into the context.

One of the ways we can use context processors in our django ecommerce application is to inject the cart items into all our pages so that no matter which page the user is on, they can see the total items in their cart.

There are a few configurations needed to make this happen. To get started, create a **context_processors.py** file in the store app. In the file, add a `cart_items` that uses the `Cart` model to retrieve cart items for the currently active user, calculates the total number of product quantities, then stores the data as shown below.

```
from .models import Cart
from django.db.models import Sum

def cart_items(request):
    if request.user.is_authenticated:
        customer = request.user
        total_cart_items = Cart.objects.filter(
            customer=customer).
           aggregate(total_quantity=Sum('quantity'))
['total_quantity'] or 0
    else:
        total_cart_items = 0
    return {'total_cart_items': total_cart_items}
```

To use the context processor, we need to add it to the templates list in the **settings.py** file.

```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR /'templates'],
        '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',
                'store.context_processors.cart_items' #new
            ],
        },
    },
]
```

Now we need to pass the data returned by the context process to the nav bar in the **base.html** file, as shown below.

```
<a class="nav-link" href="{% url 'cart-list' %}">
  <i class="fa fa-shopping-cart"></i>
  <span class="cart-count">{{total_cart_items }}</span>
</a>
```

Now if you refresh any page, you should see the total cart quantities displayed on the nav bar.

Chapter 4 : CheckOut and Payment Integration

When customers click the checkout button, we want to redirect them to a page where they enter their shipping details and pay for the order. Let's create another app called orders

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

Add the orders app to the list of installed apps in the settings.py file.

```
INSTALLED_APPS = [
# other apps
    'orders',
]
```

We also need to create the following models:

- Order
- OrderItem

Open **orders/models.py** and add an Order model class.

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

class Order(models.Model):
    customer = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE)
    phone = models.CharField(max_length=30, null=False,
blank=False)
    street_address =
models.CharField(max_length=200,null=False, blank=False )
    city = models.CharField(max_length=200,null=False,
blank=False)
    house_number =
models.CharField(max_length=200,null=False, blank=False)
    other = models.CharField(max_length=200,null=True,
blank=True)
    date_created = models.DateField(auto_now_add=True)
    date_updated = models.DateField(auto_now_add=True)
    paid = models.BooleanField(default=False)
    payment_id = models.CharField(max_length=100,null=True)
```

The Order class has the following information:

- `customer` : represents the user making the order. Using the user model, we can get the customer details such as first name, last name, email, e, etc.
- `phone`: represents the contact number of the customer.
- `street_address` is the delivery address
- `date_created`, and the `date_updated` fields represent timestamps when the order is added or updated.
- `house_number` is the customer's exact location or doorstep

- `other`: is a field where a customer can add instructions about the order.
- `Paid` is a boolean value representing the state of the order. When a customer payment is successful, we will update paid to True.
- `payment_id` represents a unique id provided by Stripe when a payment is made.

Create the `OrderItem` model class.

```
class OrderItem(models.Model):
    order = models.ForeignKey(Order,
on_delete=models.CASCADE,   related_name='order_items')
    product = models.ForeignKey(Product,
on_delete=models.CASCADE)
    quantity = models.IntegerField(default=1)
    price = models.DecimalField(
              max_digits=10,decimal_places=2, default= 0)

    def __str__(self):
        return str(self.order)

    @property
    def subtotal(self):
        return self.product.price * self.quantity
```

The OrderItem class will contain information about individual items in the order. It contains the

- `order`: the order in which the individual order item belongs to
- `product` : the product in the order
- `quantity`: the number of products in the order for each item
- `subtotal` : the total amount for the order_item

Perform migrations

```
python manage.py makemigrations
```

You should see something like this:

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

Run the migrate command

```
python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, orders,
sessions, store, users
Running migrations:
  Applying orders.0001_initial... OK
```

OrderForm

Let's create the checkout page. The checkout page will allow the customer to supply their billing information and show the total cost of the items in their order.

Start by creating the order form; in the orders app, create a **forms.py** file and add an `OrderForm` class.

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

class OrderForm(forms.ModelForm):
    class Meta:
        model = Order
        fields =  ['phone','street_address','city'
                'house_number','other']
        labels = {
            'other': 'Any other Instructions',
        }
```

Django allows us to build forms from models by using the ModelForm class. When you use the ModelForm, all the validation and data defined in the model will be inherited from the model, and you dont have to worry about validating fields.

The form class takes an inner Meta class where we define the fields which will be shown to the user. As you can see, we didn't specify the customer since it's a foreign key and will be populated by information of the currently authenticated user.

We also define a custom label for the other field.

Checkout Page

Now that we have our form let's render it. Create a `CreateCheckoutSessionView` class in **orders/views.py .**

```
from django.shortcuts import render,redirect
from django.views import View
from store.models import Product,Cart
from .forms import OrderForm

class CreateCheckoutSessionView(View):
    def get(self, request, *args, **kwargs):
        form = OrderForm()
```

```
        cart_items = Cart.objects.filter(customer =
request.user)
        total_sum = sum(item.total_cost for item in
cart_items)
        context = {'form':form,
                   'cart_items':cart_items,
                   'total_sum':total_sum
                  }
        return render(request,'orders/checkout.html',
context )
```

We first import the `Cart` model, the `OrderForm`, and the django `View` base class. Then create the `CreateCheckoutSessionView`, which inherits from the base `View` class. Inside the `CreateCheckoutSessionView`, we define a get method and perform some queries.

1. `Cart.objects.filter(customer=request.user)`: will query the database and retrieve all the Cart objects belonging to the currently authenticated user, accessed through request.user.
2. `total_sum = sum(item.total_cost for item in cart_items)`: will calculate the total sum of the total_cost attribute for each item in the `cart_items` queryset. We use `Sum`, a built-in Python function, to get the `total_sum`

We then pass the `cart_items` and the `total_sum` to the context and render the checkout page.

Create the **checkout.html** page in the templates folder.

```
.orders
└── templates
    └── orders
        └── checkout.html
```

Create a file **urls.py** in the orders app directory. Map the view to a url.

```
from django.urls import path
from . import views
urlpatterns = [

path('checkout',views.CreateCheckoutSessionView.as_view(),
name='checkout' ),

]
```

In the root **urls.py** files, include the **orders.urls**

```
urlpatterns = [

  path('admin/', admin.site.urls),
  path('',include('store.urls')),
  path('', include('orders.urls')),

]
```

In the **checkout.html** page, add the code below.

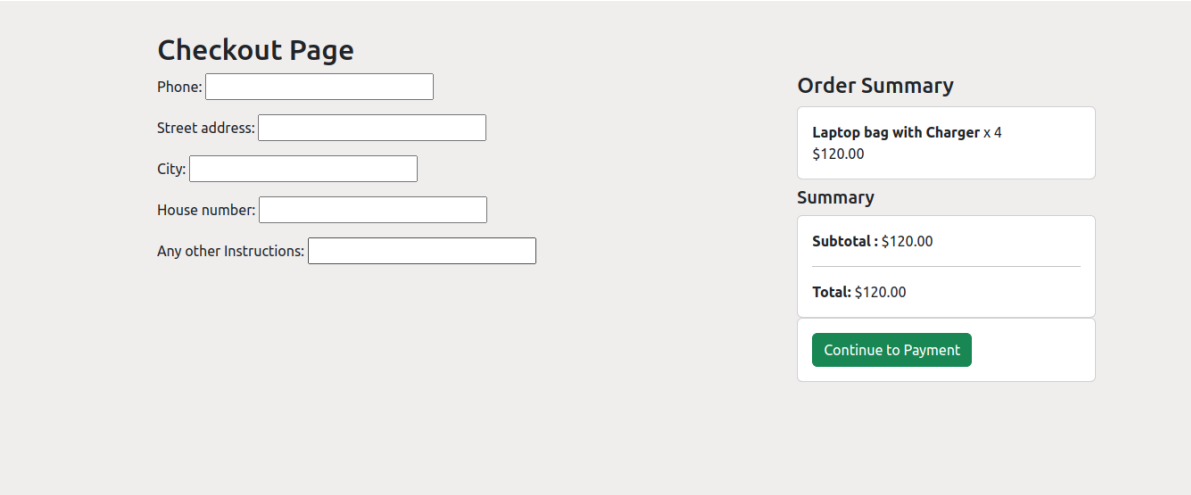
```
{% extends 'base.html' %}
{% load crispy_forms_tags %}

{% block content %}
<div class="container mt-5">
    <h2>Checkout Page</h2>
    <div class="row">
        <div class="col-md-8">
            <!-- Customer Details Form -->
            <form method="post" action=" ">
```

```
                {% csrf_token %}
                {{ form|crispy }}
        </div>
        <div class="col-md-4">
            <h4>Order Summary</h4>
            {% for item in cart_items %}
            <div class="card mb-2">
                <div class="card-body">
                    <strong>{{ item.product.name }}
</strong>
                    <span class="float-right">x {{
item.quantity }}</span>
                    <br>
                    <span>${{ item.total_cost }}</span>
                </div>
            </div>
            {% endfor %}
            <h5>Summary</h5>
            <div class="card">
                <div class="card-body">
                    <div><strong>Subtotal : </strong><span
class="float-right">${{ total_sum }}</span></div>
                    <hr>
                    <div><strong>Total: </strong><span
class="float-right">${{ total_sum }}</span></div>
                </div>
            </div>

            <div class="card">
                <div class="card-body">
                    <button type="submit" class="btn btn-
success">Continue to Payment</button>
                </div>
            </div>
        </div>
        </form>
    </div>
</div>
{% endblock %}
```

In the **store/cart_list.html** page, add the checkout url.

```
<div class ="row mt-5">
   <div class="col-sm-6">
      <a href="{% url 'checkout'%}" class="btn btn-
warning">CHECKOUT ($ {{total_cost}})</a>
   </div>
</div>
```

Now when you click the checkout button, you will be redirected to the checkout page, which looks like this:



Crispy Forms

Crispy forms are used to enhance the appearance of our django forms.

Install crispy forms with pip.

```
pip install django-crispy-forms
```

Add the crispy form to the list of installed apps in the settings.py file.

```
INSTALLED_APPS = [
 #other apps
 'crispy_forms',
]
```

Install bootstrap5 and add the template pack to the installed apps.

```
pip install crispy-bootstrap5
```

Add the Bootstrap5 template pack to installed apps.

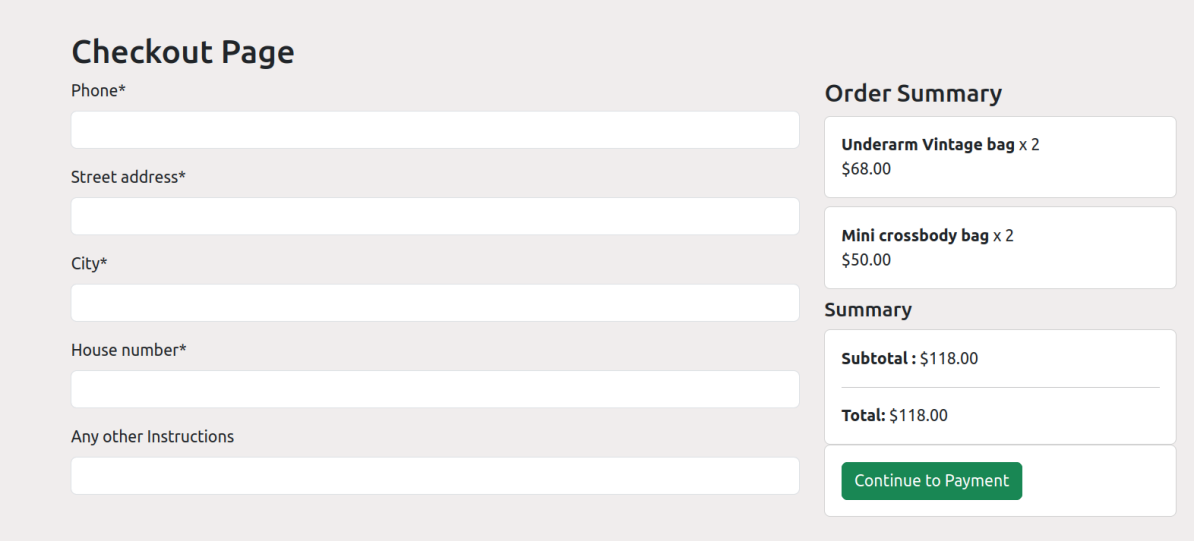
```
INSTALLED_APPS = [
 #other apps
'crispy_forms',
 'crispy_bootstrap5',
]
```

Add this setting in the settings.py file.

```
 CRISPY_TEMPLATE_PACK = 'bootstrap5'
```

Add the crispy form filter and the `{% crispy %}` tag in the **checkout.html** file. Refresh the page, and now the form looks much better.

Our checkout page now looks better.

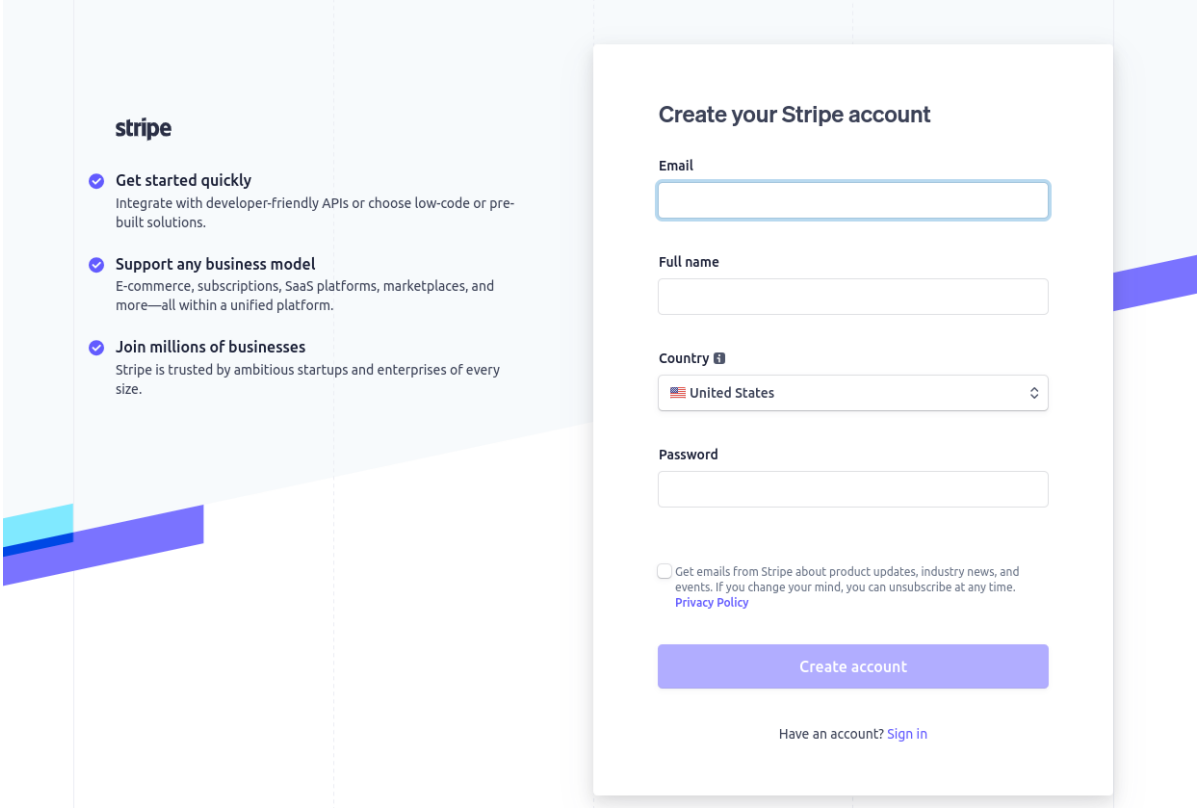


As you can see, we have a summary of the number of products in the cart, their subtotals, and the totals.

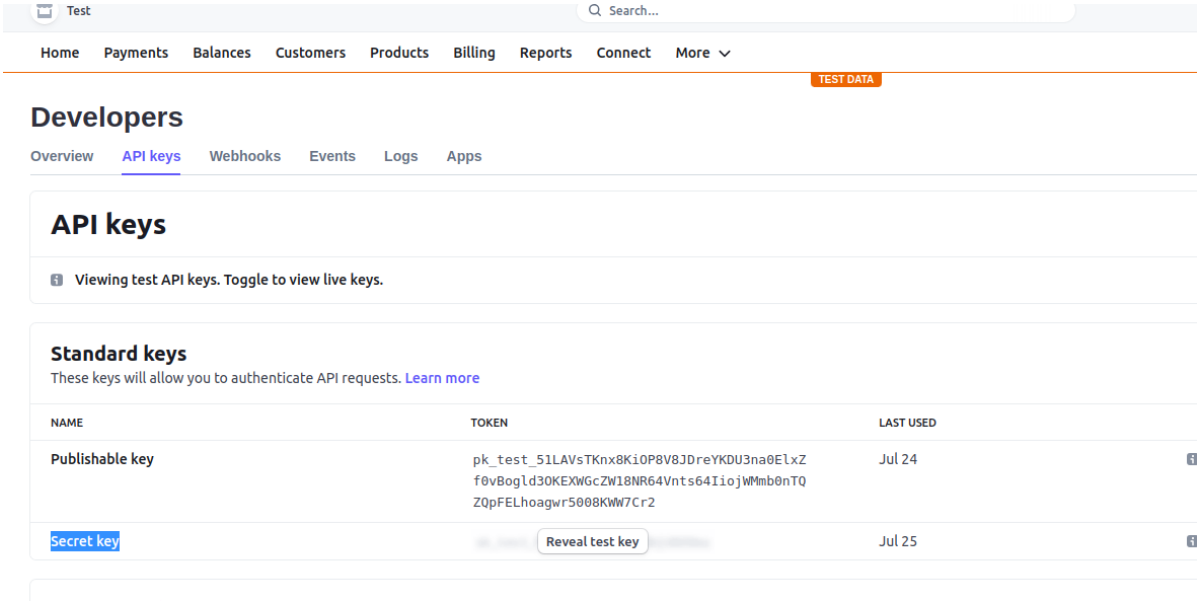
The Continue to Payment button should redirect the user to a stripe page. It doesn't do anything since we haven't created the logic for the POST method. We need to integrate our application with Stripe. In the next section, we will look at Stripe.

Stripe Integration

To use Stripe, you need a Stripe account, head over to https://dashboard.stripe.com/register and register.



After successfully registering an account, head over to https://dashboard.stripe.com/, and on the API section, you will get your API keys.



Install the Python stripe API package with pip.

```
pip install stripe
```

Stripe provides 2 types of API keys:

- Publishable key
- Secret key

Copy the API keys and add them to the settings.py file.

```
DOMAIN_ROOT = "http://127.0.0.1:8000/"
STRIPE_PUBLISHABLE_KEY ='Your Publishable key '
STRIPE_SECRET_KEY = 'your Secret key'
```

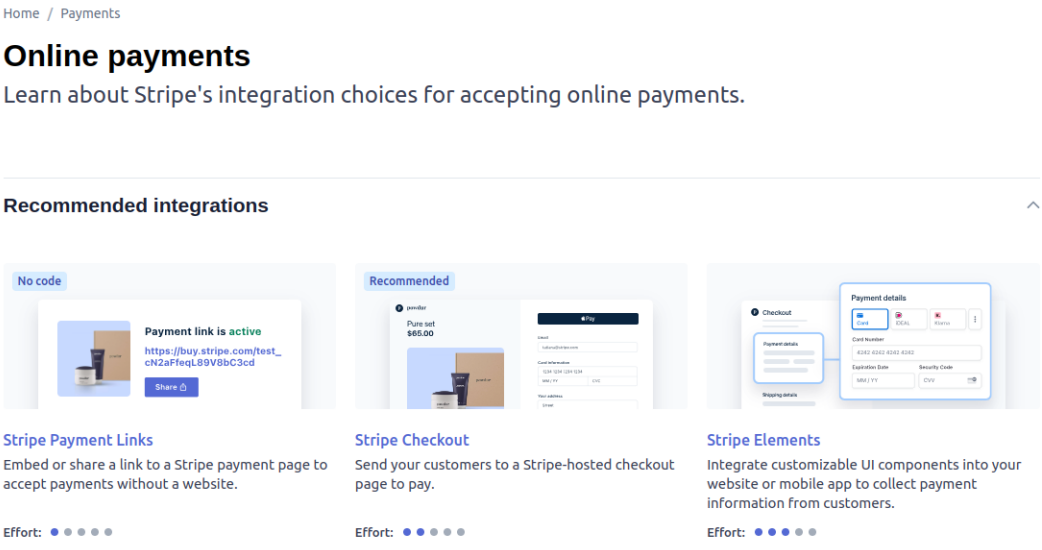
Ensure the keys are defined as strings. DOMAIN_ROOT is the root endpoint where your django application is running.

In your base.html, add the stripe js script between the head tags.

```
<script src="https://js.stripe.com/v3/"></script>
```

Stripe provides 3 types of payment integrations.

- Stripe Payment Links
- Stripe Checkout
- Stripe Elements



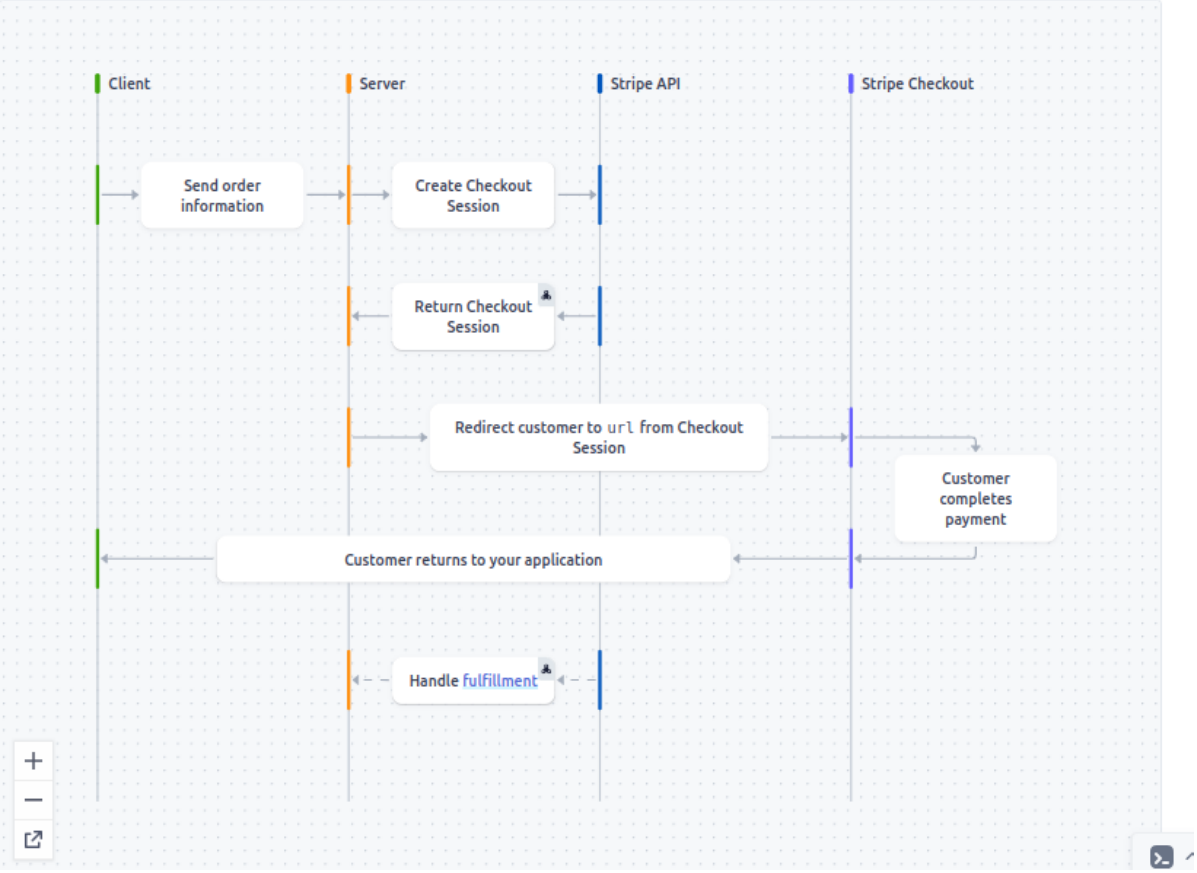
We will use the Stripe checkout integration, which provides a Stripe-hosted checkout page for customers to pay.

How Stripe Checkout Works

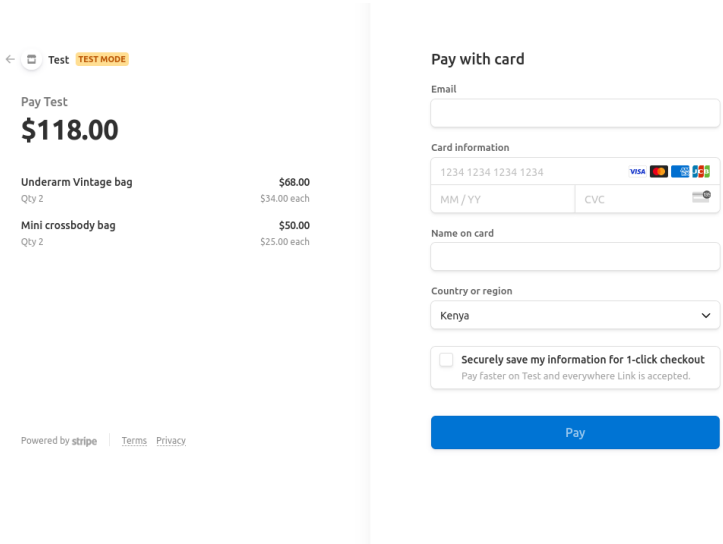
The Stripe checkout process looks like this:

1. When customers are ready to complete their purchase, your application creates a new Checkout Session.
2. The Checkout Session provides a URL that redirects customers to a Stripe-hosted payment page.
3. Customers enter their payment details on the payment page and complete the transaction.
4. After the transaction, a webhook fulfills the order using the checkout.session.completed event.

Here is a Screenshot from the Stripe documentation that shows how Stripe Checkout works.



The Stripe-hosted payment page looks like this.



The Stripe checkout page also requires a success_url page where customers are redirected after payment is successful and a cancel_url page where customers are redirected if the payment is unsuccessful.

Stripe Checkout Example

The Stripe Checkout API takes a few attributes, the most common are:

1. **line_items**: The list of items the customer is purchasing
2. **client_reference_id**: A unique string used to reference the Checkout Session; it can help to reconcile the Session with your internal systems.
3. **currency**: Three-letter ISO currency code
4. **customer_email**: Prefilled customer email address for the payment flow. If not set, customers will be asked to provide an email address.
5. **metadata**: A set of key-value pairs that you can attach additional information, for example, the order_id
6. **mode**: The mode of the Checkout Session can be payment setup or subscription
7. **success_url**: A URL where the customer will be redirected after payment or subscription is successful.
8. **cancel_url**: A URL where customers are directed if they cancel the payment and return to your website.

9. **url**: URL to the Checkout Session, used to redirect customers to Checkout.; This is a stripe-hosted url.

A simple Stripe checkout would look like this:

```
import stripe
stripe.api_key = 'api_key'

domain_root_url = 'example.com'
def create_checkout_session():
    try:
        checkout_session = stripe.checkout.Session.create(
            line_items=[
                {
                    'name': 'pillow',
                    'price': '2000',
                    'quantity': 1,
                },
            ],
            mode='payment',
            success_url=domain_root + '/success.html',
            cancel_url=domain_root + '/cancel.html',
        )
    except Exception as e:
        return str(e)

    return redirect(checkout_session.url, code=303)
```

First, we import the stripe library, then authenticate to the stripe API with `stripe.api` .`stripe.checkout.Session.create()` creates a checkout session and defines the required attributes .i.e : `line_items`, `mode of payment`,`success_url`, and `cancel_url` .Stripe uses the smallest unit of price, i.e., cents, so the prices are converted to cents. For example, $20 will be 2000 cents

If an exception occurs, it is caught by the except block.

Finally, we redirect the user to the checkout URL (a stripe-hosted page) where they will make the payment.

Now that we know how the Stripe checkout works, we can add the functionality in our django application.

Let's first define the success and cancel pages. In **orders/views.py,** add the code below.

```
def success_page(request):
    return render(request, 'success.html')


def cancel_page(request):
    return render(request, 'cancel.html')
```

Add success.html and cancel.html pages to the root templates folder next to the base.html file.

```
.templates
├── base.html
├── cancel.html
└── success.html
```

success.html

```
{% extends 'home.html' %}
{% block content %}

<div class="container mt-5">
    <div class="row justify-content-center">
        <div class="col-md-6 text-center">
            <h1>Payment Successful</h1>
            <p>Thank you for your purchase! Your payment
was successful.</p>
            <a href="#" class="btn btn-primary">Go to
Orders</a>
        </div>
    </div>
```

```
</div>

{% endblock %}
```

cancel.html

```
{% extends 'home.html' %}
{% block content %}

<div class="container mt-5">
    <div class="row justify-content-center">
        <div class="col-md-6 text-center">
            <h1>Payment Unsuccessful</h1>
            <p>Your payment was cancelled or unsuccessful.
</p>
            <a href="https://localhost:8000/" class="btn
btn-primary">Go Back to Cart</a>
        </div>
    </div>
</div>

{% endblock %}
```

Map the views to urls. In the **orders/urls.py** file, update as follows.

```
urlpatterns = [
    path('success',views.success_page, name='success' ),
    path('cancel',views.cancel_page, name='cancel' ),

]
```

Create Order and Make Payment

Let's create the functionality for customers to make and pay for orders.
In **orders/views.py**, add the following imports at the top of the file.

```
from django.views.decorators.csrf import csrf_exempt
from django.http.response import JsonResponse
from django.conf import settings
from .models import OrderItem
import stripe
```

Add a post method to the `CreateCheckoutSessionView`. with a `@csrf_exempt` decorator. The `@csrf_exempt` is a function that exempts a function from Cross-site Forgery (CSRF). This step is important because we want to allow communication with the Stripe API without providing a CSRF Token.

```
class CreateCheckoutSessionView(View):
    def get(self, request, *args, **kwargs):
        form = OrderForm()
        cart_items = Cart.objects.filter(
            customer = request.user)
        total_sum = sum(
            item.total_cost for item in cart_items)
        context = {'form':form, 'cart_items':cart_items,
                   'total_sum':total_sum}
        return render(
            request,'orders/checkout.html', context )

    @csrf_exempt
    def post(self, request, *args, **kwargs):
        pass
```

In the post request, we want to perform the following:

- get the customer's shipping information
- Create an order for the customer
- create Order items from the cart and save them to the database

- Process payment through Stripe.

Update the post () method as shown below.

```
class CreateCheckoutSessionView(View):
    def get(self, request, *args, **kwargs):
        # the rest of the code here

    @csrf_exempt
    def post(self, request, *args, **kwargs):
        cart_items = Cart.objects.filter(customer =
request.user)
        form = OrderForm(request.POST)
        if form.is_valid():
                order = form.save(commit=False)
                order.customer = request.user
                order.save()

                for cart_item in cart_items:
                    OrderItem.objects.create(
                        order=order,
                        product=cart_item.product,
                        quantity=cart_item.quantity,
                        price=cart_item.product.price,

                    )
            # stripe processing


        return JsonResponse({'success':'order created'})
```

Lest breakddown what is haeppenfing.

- `cart_items = Cart.objects.filter(customer = request.user)` - retrieves the cart items for the currently authenticated user.
- `form = OrderForm(request.POST)` initializes the `OrderForm` with the data from the POST request. Then we check if the form is valid with `form.is_valid()`:

- If the form is valid, we create an order instance. By setting `form.save(commit=False)`, the order is not yet saved to the database. We then associate the order with the currently authenticated user and save the order to the database.
- Then for each of the cart items, we create an OrderItem instance. Each OrderItem is associated with the order we just created.
- Even though the order instance has been created, it's still marked as unpaid; once the payment has succeeded, we will update the order status to paid.

Now we can implement the payment process.

The line items list required by Stripe will come from the cart; let's loop through the cart items and append them to the `line_items` list.

```
line_items =[]
        for  item in cart_items:
            line_items.append({
                'name':item.product.name,
                'quantity':item.quantity,
                'amount':int(item.product.price *100),
                'currency': 'usd',

            })
```

Next, create the stripe checkout session, and specify all the required attributes. The `CreateCheckoutSessionView` should now look like this:

```
class CreateCheckoutSessionView(View):
    def get(self, request, *args, **kwargs):
        form = OrderForm()
        cart_items = Cart.objects.filter(
                    customer = request.user
                )
        total_sum = sum(
            item.total_cost for item in cart_items
```

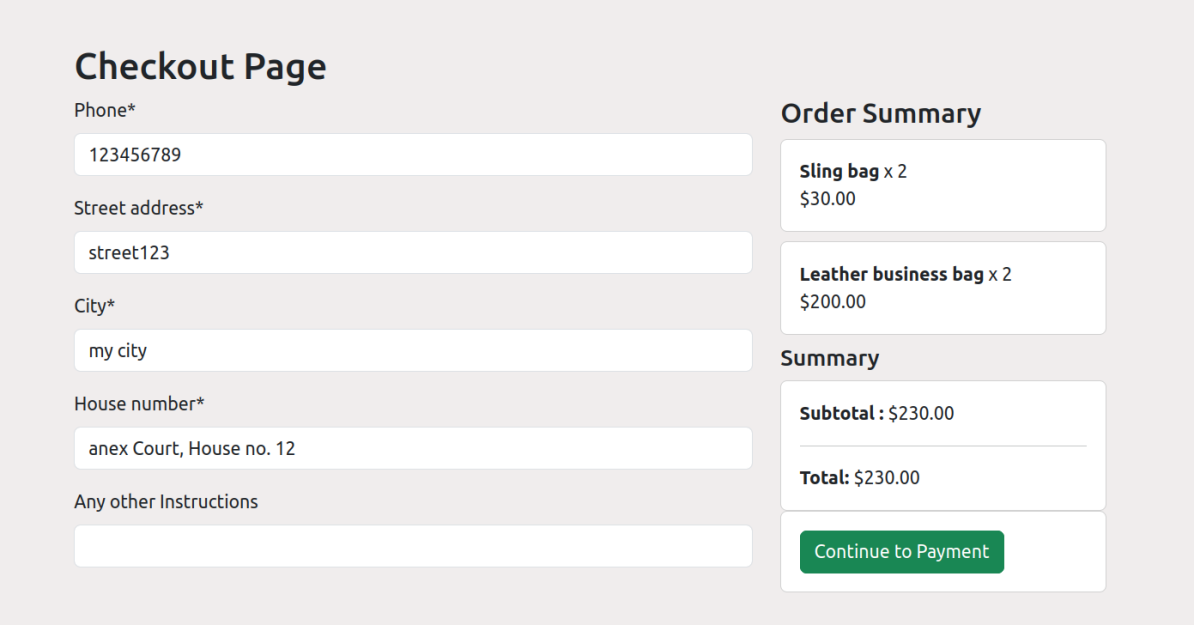
```
            )
        context = {'form':form,
                   'cart_items':cart_items,
                   'total_sum':total_sum
                  }
        return render(
            request,'orders/checkout.html',
            context )

    @csrf_exempt
    def post(self, request, *args, **kwargs):
        form = OrderForm()
        cart_items = Cart.objects.filter(
                    customer = request.user
                   )
        form = OrderForm(request.POST)
        if form.is_valid():
                order = form.save(commit=False)
                order.customer = request.user
                order.save()

                for cart_item in cart_items:
                    OrderItem.objects.create(
                        order=order,
                        product=cart_item.product,
                        quantity=cart_item.quantity,
                        price=cart_item.product.price,

                    )

        line_items =[]
        for  item in cart_items:
            line_items.append({
                'name':item.product.name,
                'quantity':item.quantity,
                'amount':int(item.product.price *100),
                'currency': 'usd',

            })
```

```
        domain_url = settings. DOMAIN_ROOT
        stripe.api_key = settings.STRIPE_SECRET_KEY

        try:
            checkout_session =
stripe.checkout.Session.create(
            client_reference_id=request.user.id,
            metadata = {'order_id':order.id},
            customer_email= request.user.email,
            success_url =domain_url+ 'success?session_id=
{CHECKOUT_SESSION_ID}',
            cancel_url=domain_url + 'cancel/',
            payment_method_types=['card'],
            mode='payment',
            line_items = line_items,

            )

            request.session['checkout_session_id'] =
checkout_session['id']
            return redirect(checkout_session.url, code=303)
        except Exception as e:
            return JsonResponse({'error':str(e)})
```

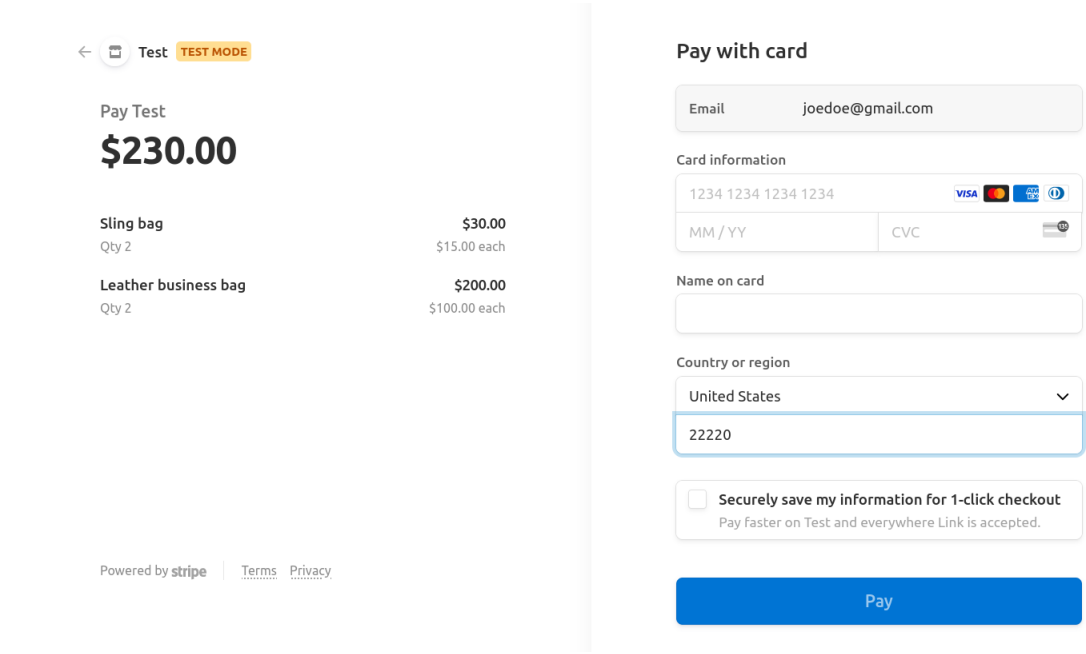
- `stripe.api_key = settings.STRIPE_SECRET_KEY` authenticates to the Stripe API.
- `stripe.checkout.Session.create` creates a checkout session containing the required attributes. For the `client_reference_id` we specify the current user, the `metadata` contains the `order_id`, and the `customer_email` is the email of the currently authenticated user.

After a checkout session is created, the user is redirected to the session url, a payment page showing the line items, the order amount, and the currency. If any error occurs, it will be caught in the except block.

In the http://127.0.0.1:8000/checkout page, add the shipping details and click the Continue to Payment button.



You should be redirected to a page that looks like this:

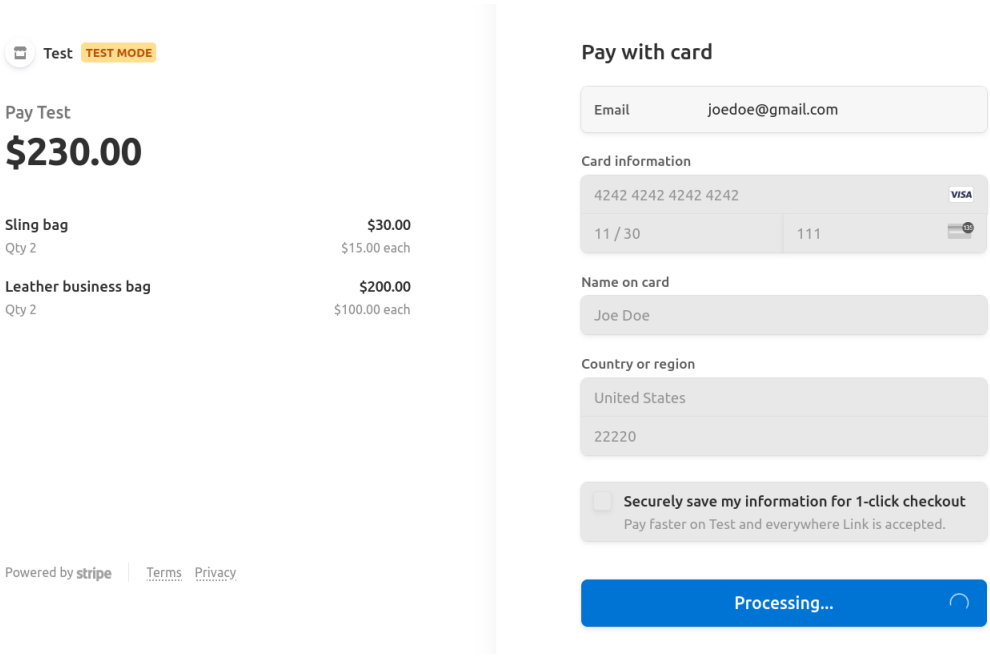


As you can see, the customer's email has been prefilled since we added it as an attribute in the checkout session.

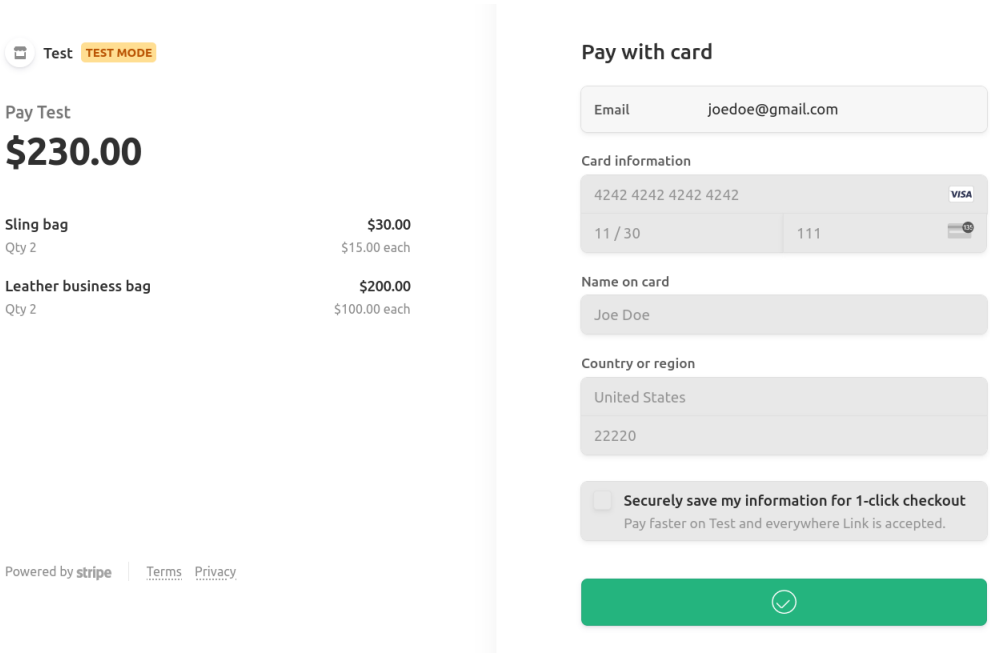
Stripe provides credit card numbers for testing purposes .

- Payment succeeds - 4242 4242 4242 4242
- Payment requires authentication - 4000 0025 0000 3155
- Payment is declined - 4000 0025 0000 3155

After clicking the Pay button, you should see the payment processing progress bar.

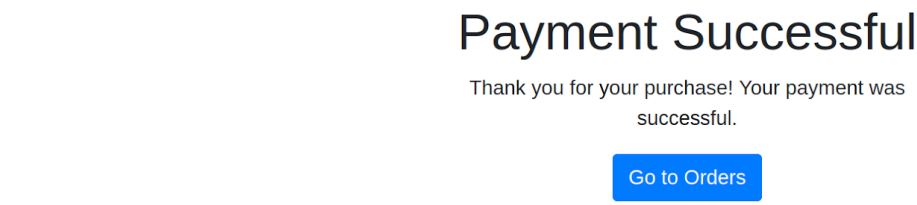


When the payment is successful, the button will turn green.

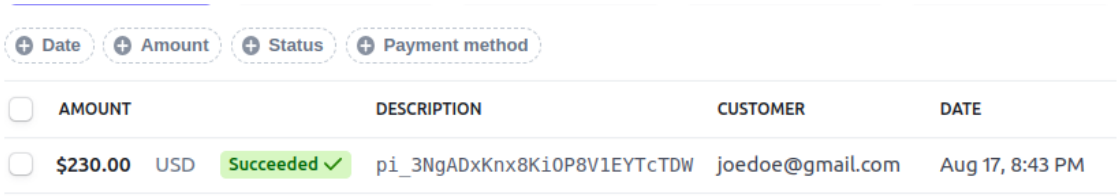


Then you will be redirected to the success page

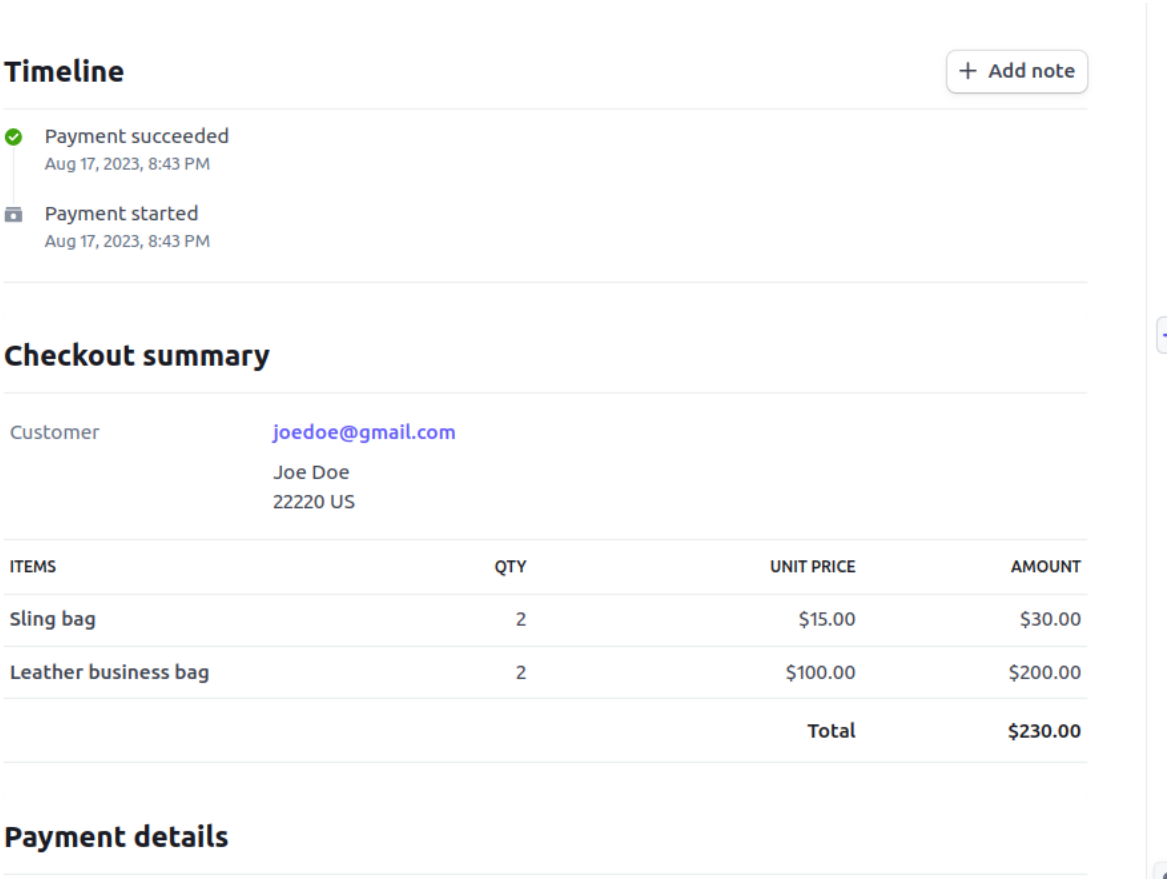
s



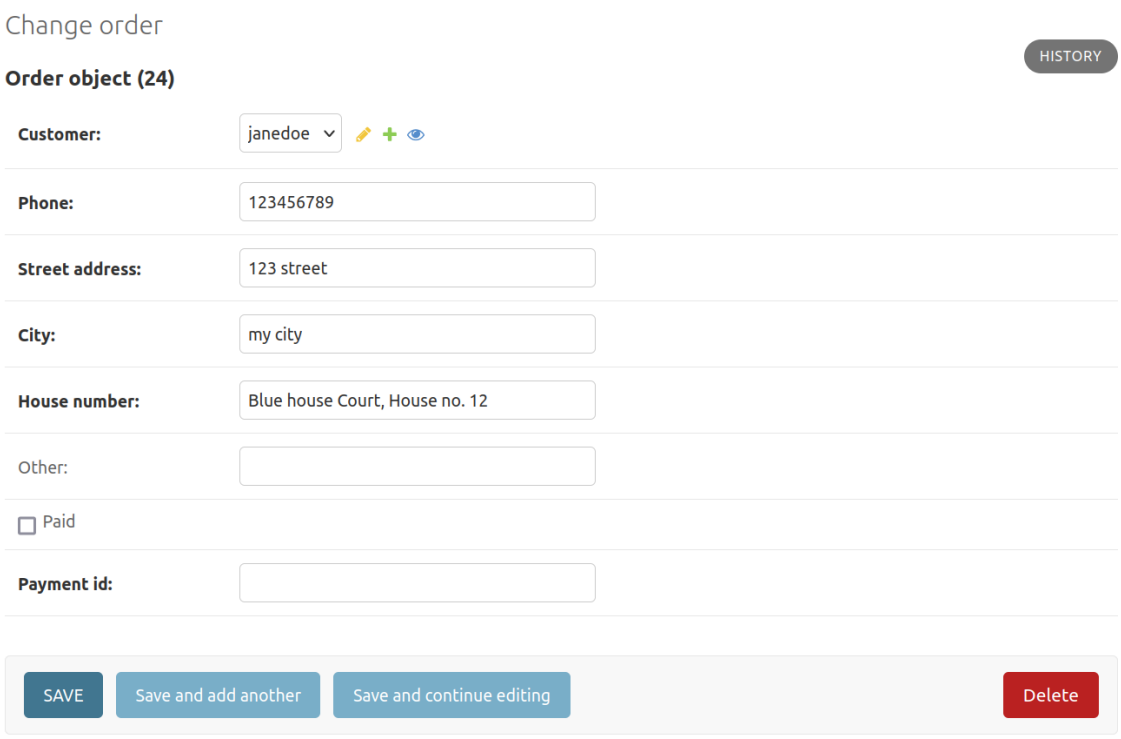
In your stripe dashboard, you can see the payment has succeeded.



When you expand the payment, you should see all the details the customer provided and the Checkout summary.



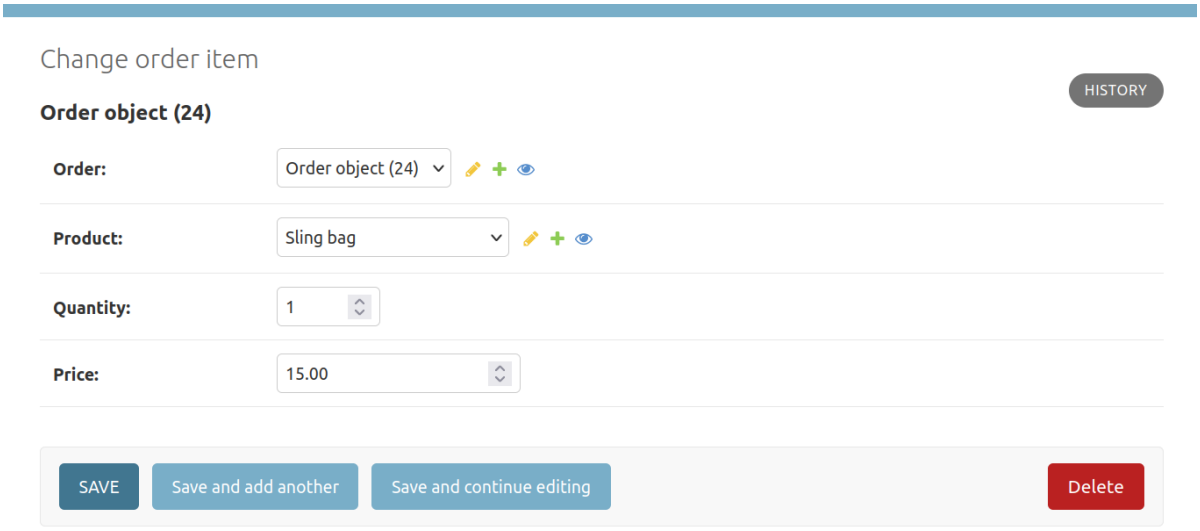
If you go to the admin site, the order instance looks like this:



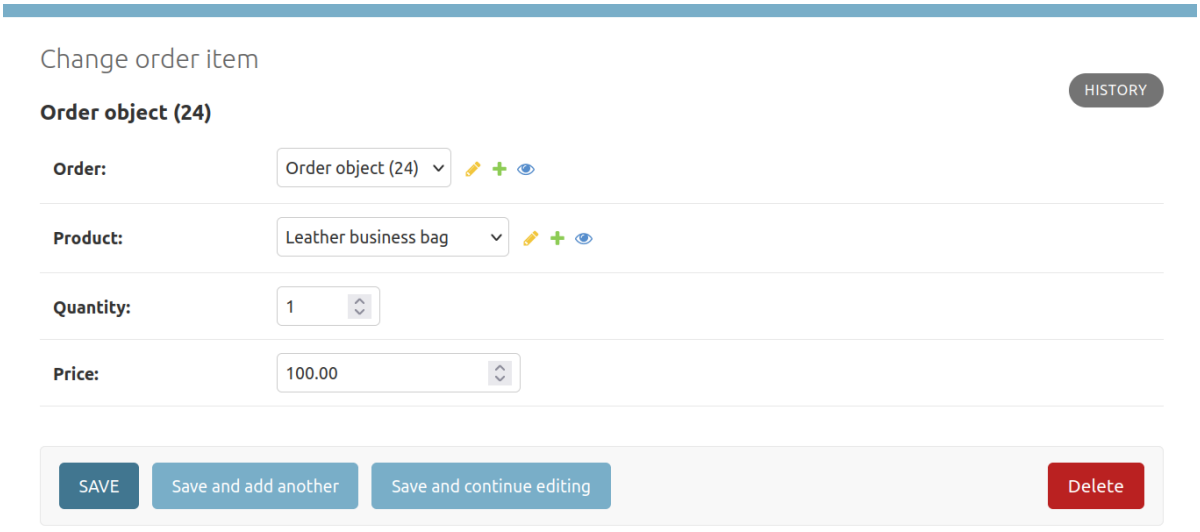
As you can see, the order has not been set as paid, and there is still no payment id. We will use Stripe Webhooks to confirm the payment and update the order details.

On the orderItem records, the order items have been created.

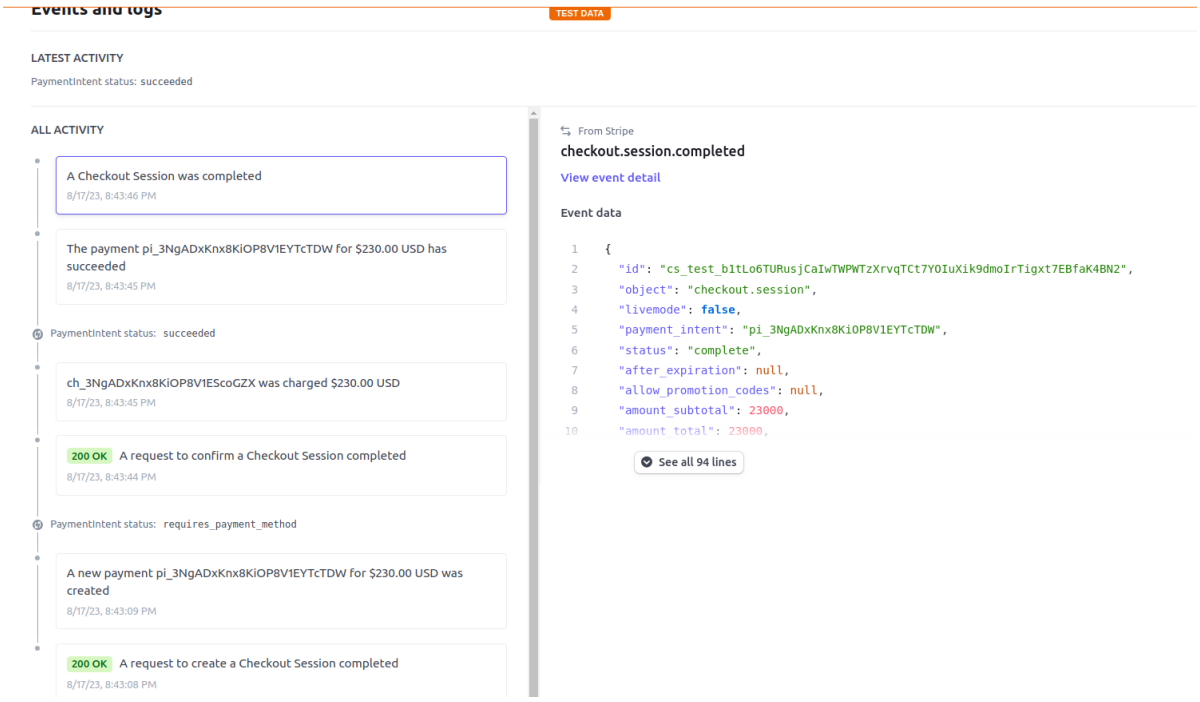
first order item



second order item



Stripe also records a log of each event for every transaction.



Stripe WebHooks

Stripe uses Webhooks to provide real-time updates from your Stripe account to your application. Webhooks are essential for providing real-time updates about events happening outside your application.

For example, when a customer pays for a product, Stripe Webhooks let you know that a payment has happened; This allows your application to automate processes and synchronize payment status between your application and Stripe.

For a webhook to happen, we need the following:

- Webhook endpoint - This is an endpoint in your application that Stripe can send requests to.
- Webhook event - This is the data that Stripe sends to the endpoint.

Stripe has a variety of events, which you can find out more about here. However, in our application, we are interested in these events:

- `checkout.session.completed`: Triggered when a customer completes the checkout process.

- `checkout.session.payment_failed`: Triggered when a payment fails during checkout

We will use the Stripe CLI, a command line interface provided by Stripe, for testing webhook endpoints locally.

Stripe CLI

Head over to the https://stripe.com/docs/stripe-cli#install and install the Stripe CLI according to your operating system.

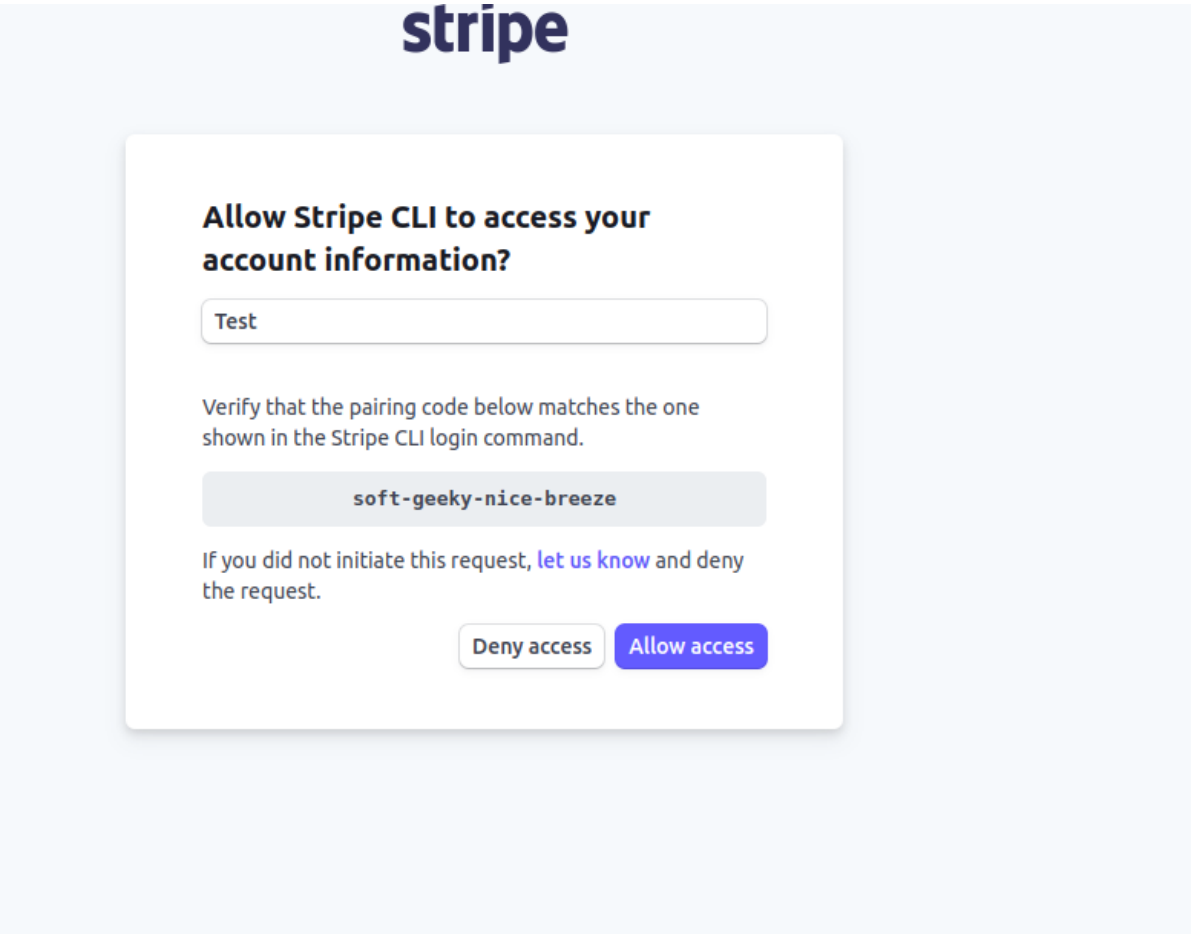
Login and authenticate with your Stripe Account.

```
stripe login
```

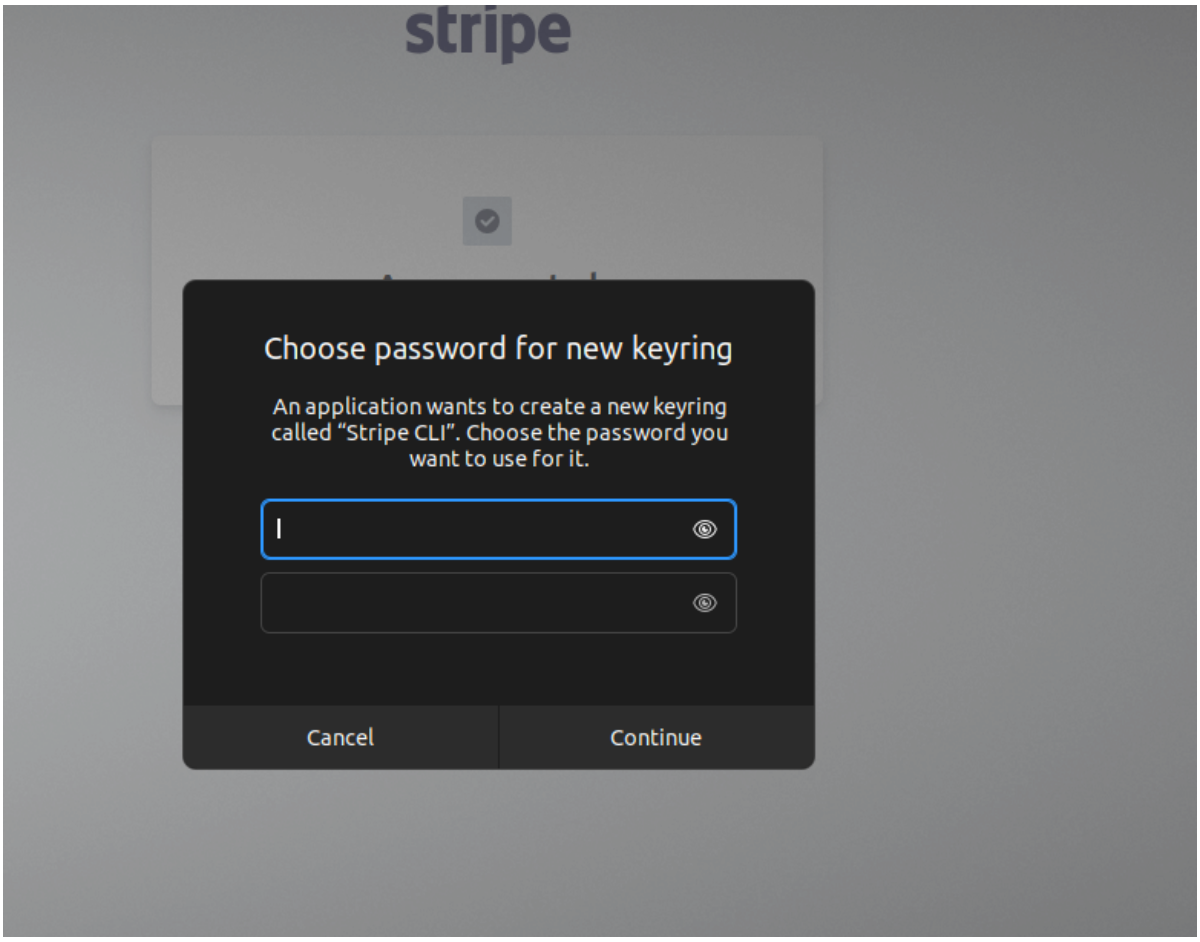
You should see something like this on your treminal

```
Your pairing code is: **********
This pairing code verifies your authentication with Stripe.
Press Enter to open the browser or visit
https://dashboard.stripe.com/stripecli/confirm_auth?
t=uvOoghOycm3T4wdnnVzajUmVnntdfSIO (^C to quit)
```

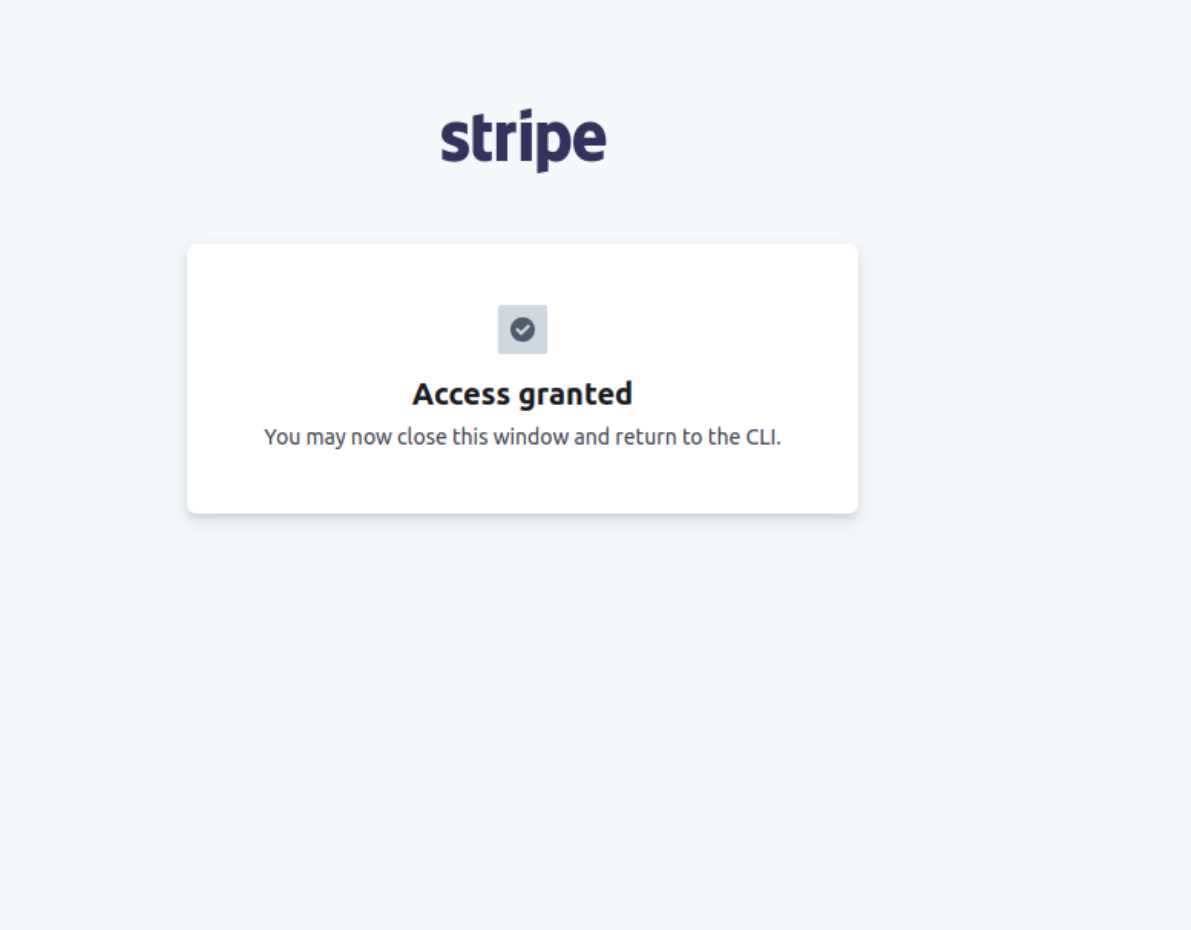
When you press enter, you will be redirected to this page requesting access.



When you allow access, you will be redirected to this page,



Set a the password you want to use and click continue.



Now you have granted stipe access to listen to events.

Webhook Endpoint

Lets first create the web hook. Add the endpoint secret key in the settings.py file.

```
STRIPE_ENDPOINT_SECRET = 'your_endpoint_secret'
```

```
@csrf_exempt
def stripe_webhook(request):
    stripe.api_key = settings.STRIPE_SECRET_KEY
    endpoint_secret = settings.STRIPE_ENDPOINT_SECRET
    payload= request.body
```

```
    event = None
    sig_header = request.headers.get('stripe-signature')
    try:
        event = stripe.Webhook.construct_event(
            payload, sig_header, endpoint_secret
        )
    except stripe.error.SignatureVerificationError as e:
        return HttpResponse(status =400)

    if event['type'] == 'checkout.session.completed':
        payment_intent = event['data']['object']
        print(payment_intent)
    return HttpResponse(status=200)
```

The stripe_webhook function does the following

- `payload= request.body` - retrieves the payload data from the request's body,
- `sig_header = request.headers.get('stripe-signature')` - verifies STRIPE SIGNATURE
- `event = stripe.Webhook.construct_event(payload, sig_header, endpoint_secret )` : constsructs a webhook event object using Stripe's `stripe.Webhook.construct_event ()` method, The `stripe.Webhook.construct_event ()` method takes the `payload`,`sig_header` and `endpoint_secret` as paramaters.

If any errors occur, they are caught in the exception blocks and if the event type is checkout.session.completed, we print the event object.

Map the view to a url.

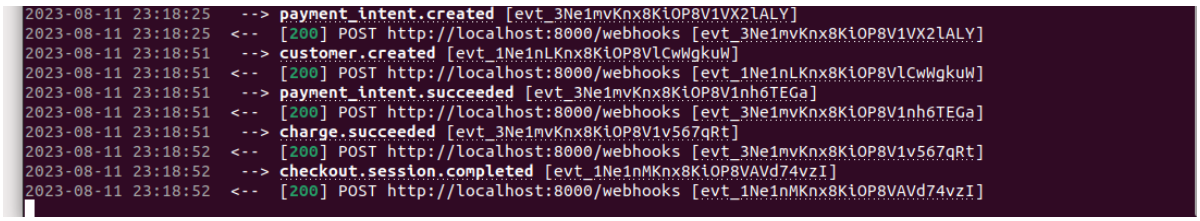
```
urlpatterns = [

    path('checkout',views.create_checkout_session,
name='checkout' ),
    path('success',views.success_page, name='success' ),
    path('cancel',views.cancelled_page, name='cancel' ),
    path('webhooks',views.stripe_webhook, name='webhooks'
), #new

]
```

In a separate terminal, issue this command to listen to payment events.

```
stripe listen --forward-to localhost:8000/webhooks
```

where localhost:8000/webhooks is the url of the webhook.

Now if you make another payment, you should see this output on the terminal listening to the webhook events.



The event object received looks like this:

```
{
  "after_expiration": null,
  "allow_promotion_codes": null,
  "amount_subtotal": 24000,
  "amount_total": 24000,
  "automatic_tax": {
    "enabled": false,
    "status": null
```

```
  },
  "billing_address_collection": null,
  "cancel_url": "http://127.0.0.1:8000/cancel/",
  "client_reference_id": "2",
  "consent": null,
  "consent_collection": null,
  "created": 1692790884,
  "currency": "usd",
  "currency_conversion": null,
  "custom_fields": [],
  "custom_text": {
    "shipping_address": null,
    "submit": null
  },
  "customer": "cus_OVFxUSZUOpuDTY",
  "customer_creation": "always",
  "customer_details": {
    "address": {
      "city": null,
      "country": "US",
      "line1": null,
      "line2": null,
      "postal_code": "22220",
      "state": null
    },
    "email": "joedoe@gmail.com",
    "name": "django",
    "phone": null,
    "tax_exempt": "none",
    "tax_ids": []
  },
  "customer_email": "joedoe@gmail.com",
  "expires_at": 1692877284,
  "id":
"cs_test_a1m8Rl6DTZg5GWOJLOixaGrlTqtNnci7pnVZY3aKktZBYOOY4z
IVmpYUjl",
  "invoice": null,
  "invoice_creation": {
    "enabled": false,
    "invoice_data": {
```

```
      "account_tax_ids": null,
      "custom_fields": null,
      "description": null,
      "footer": null,
      "metadata": {},
      "rendering_options": null
    }
  },
  "livemode": false,
  "locale": null,
  "metadata": {
    "order_id": "43"
  },
  "mode": "payment",
  "object": "checkout.session",
  "payment_intent": "pi_3NiFRAKnx8KiOP8V0wIwEF21",
  "payment_link": null,
  "payment_method_collection": "always",
  "payment_method_options": {},
  "payment_method_types": [
    "card"
  ],
  "payment_status": "paid",
  "phone_number_collection": {
    "enabled": false
  },
  "recovered_from": null,
  "setup_intent": null,
  "shipping": null,
  "shipping_address_collection": null,
  "shipping_options": [],
  "shipping_rate": null,
  "status": "complete",
  "submit_type": null,
  "subscription": null,
  "success_url": "http://127.0.0.1:8000/success?session_id=
{CHECKOUT_SESSION_ID}",
  "total_details": {
    "amount_discount": 0,
    "amount_shipping": 0,
```

```
    "amount_tax": 0
  },
  "url": null
}
```

As you can see from the event response object, we can get the following information about the order:

- `user_id = payment_intent['client_reference_id']`
- `order_id = payment_intent['metadata']['order_id']`
- `amount_total_dollars = payment_intent['amount_total'] / 100`

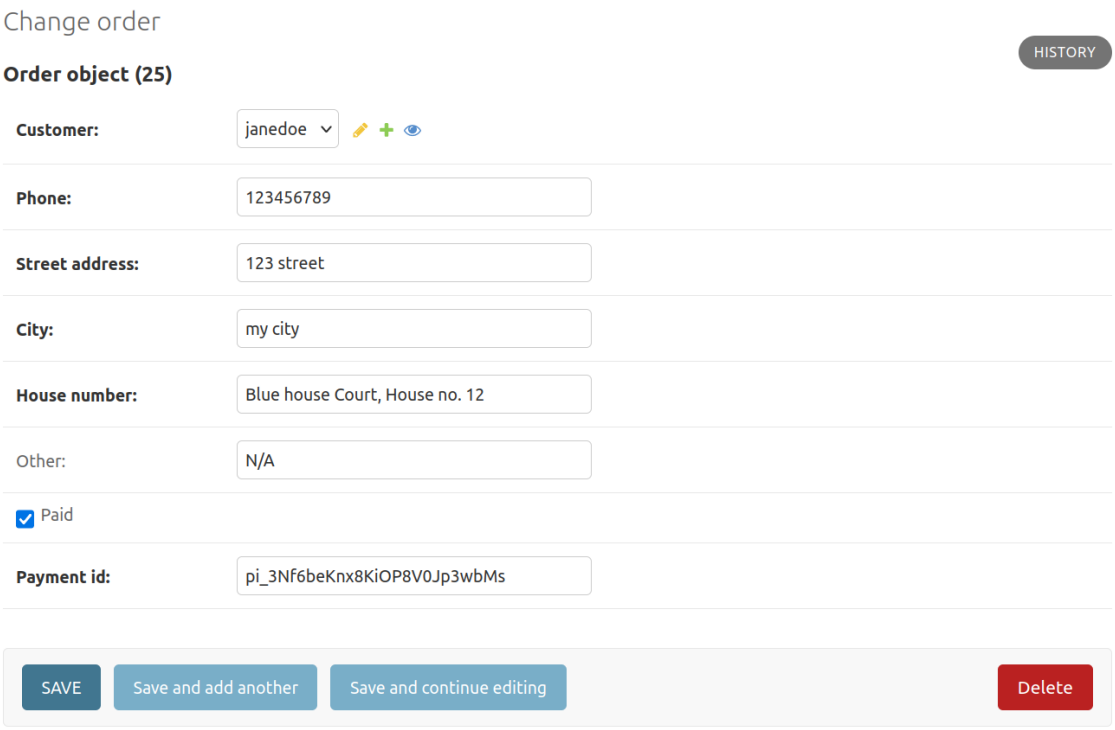
Now, we can update the order status to Paid for the particular user, email the user, and clear items in the cart.

Update the `stripe_webhook()` as follows:

```
@csrf_exempt
def stripe_webhook(request):
    stripe.api_key = settings.STRIPE_SECRET_KEY
    endpoint_secret = settings.STRIPE_ENDPOINT_SECRET
    payload= request.body
    event = None
    sig_header = request.headers.get('stripe-signature')
    try:
        event = stripe.Webhook.construct_event(
            payload, sig_header, endpoint_secret
        )
    except stripe.error.SignatureVerificationError as e:
        return HttpResponse(status =400)
    # payment_intent.succeeded'
    if event['type'] == 'checkout.session.completed':
        payment_intent = event['data']['object']
        print(payment_intent)
```

```
        user_id = payment_intent['client_reference_id']
        order_id = payment_intent['metadata']['order_id']

        order =  get_object_or_404(Order, id = order_id)
        order.payment_id = payment_intent['payment_intent']
        order.paid =True
        amount_total_dollars =
payment_intent['amount_total'] / 100
        order.amount = amount_total_dollars
        order.save()


        user = User.objects.get(id =user_id)
        Cart.objects.filter(customer=user).delete()


        # TO DO : send email


    return HttpResponse(status=200)
```

Now if you check the status for this particular order, you can see that it has a payment id and the paid status is set to true.



Emails

Sending emails is a critical component of any application. The password reset process relies on emails to ensure users successfully reset their passwords. Emails can also send users notifications and other important information to admins.

Django provides a built-in email module `django.core.mail`, that allows us to send emails from our applications.

A quick way to send emails in django looks like this:

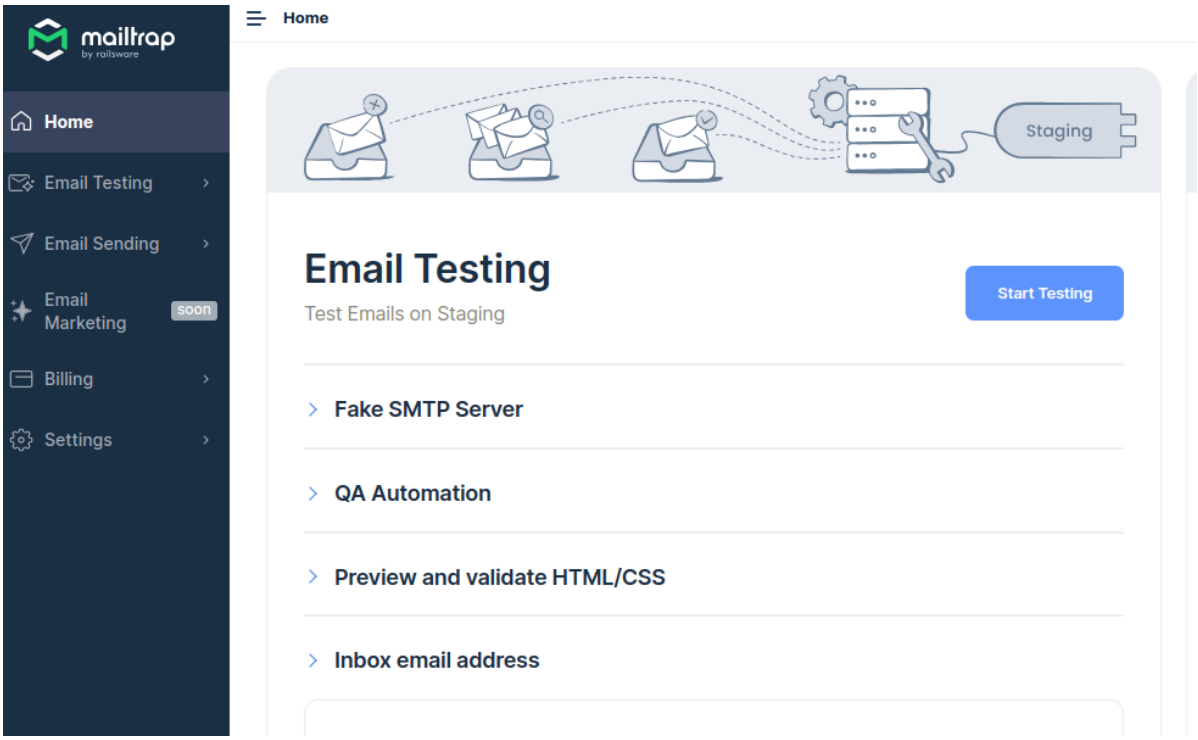
```
from django.core.mail import send_mail

send_mail(
    "Subject here",
    "Here is the message.",
    "from@example.com",
    ["to@example.com"],
    fail_silently=False,
)
```

We need to configure email settings in our django application. These settings are:

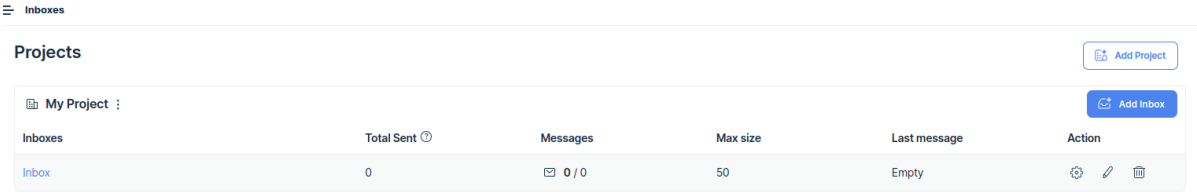
```
# settings.py
EMAIL_BACKEND =
'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'your_smtp_host' # your SMTP host
EMAIL_PORT = 587 # SMTP port
EMAIL_USE_TLS = True # Set to True if your SMTP server uses
TLS
EMAIL_USE_SSL = False # Set to True if your SMTP server
uses SSL
EMAIL_HOST_USER = 'your_email@example.com'
EMAIL_HOST_PASSWORD = 'your_email_password'
DEFAULT_FROM_EMAIL = 'your_email@example.com'
```

Send Emails with MailTrap

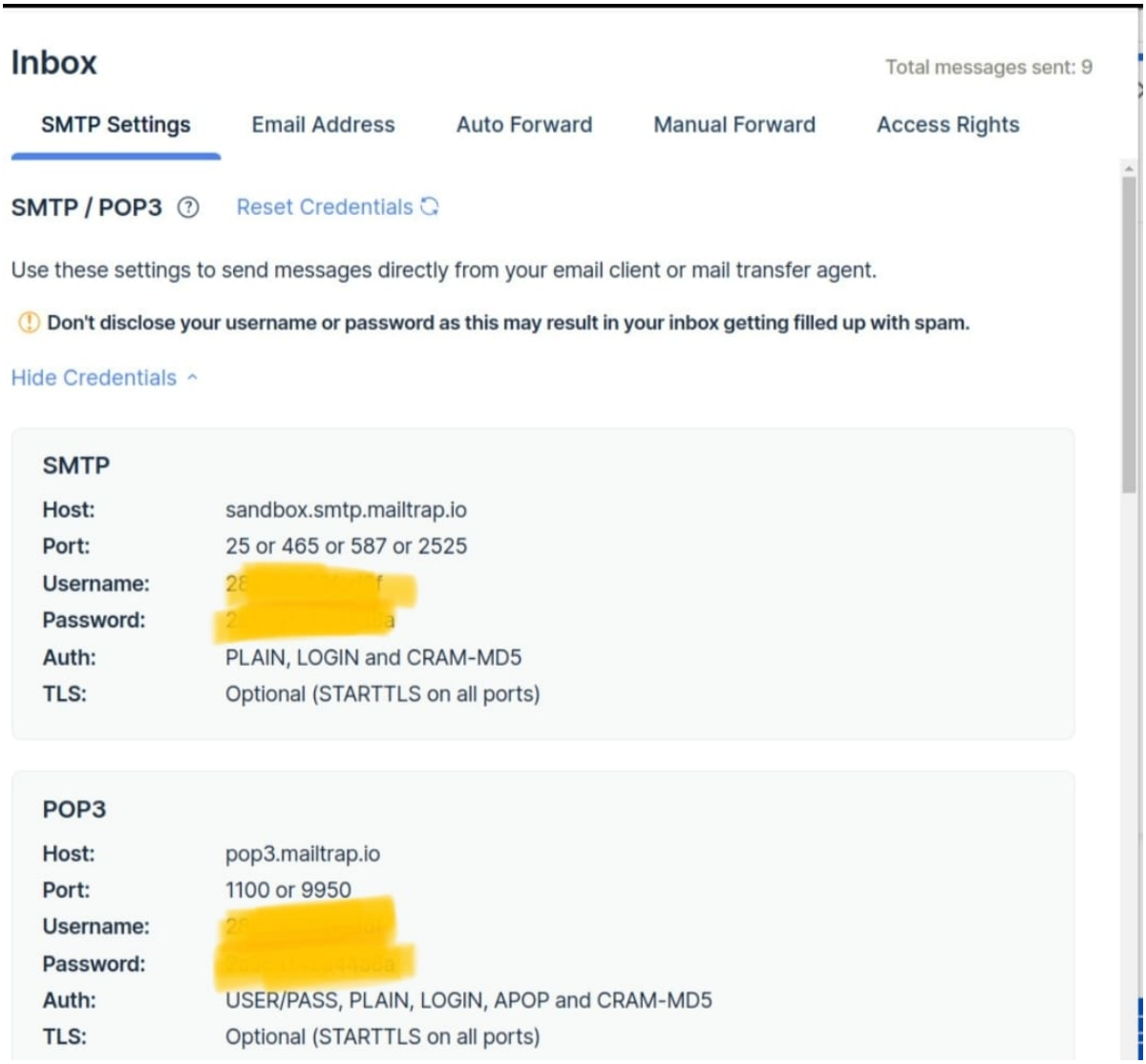
Head over to https://mailtrap.io/ and set up a free Mailtrap account. Once set up, you should see the dashboard, which looks like this:



Click on the default inbox.



You should see this page, which shows your SMTP settings. In your settings.py file, define the email configurations as shown below.




```
EMAIL_BACKEND =
'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'sandbox.smtp.mailtrap.io'
EMAIL_PORT = '587'
EMAIL_HOST_USER = 'mailtrap_user'
EMAIL_HOST_PASSWORD = 'mailtrap_password'
EMAIL_USE_TLS = True
EMAIL_USE_SSL = False
```

Your django application is now configured to send emails. It's important to note that emails sent in the testing mode will not appear in the inbox but will be shown in your Mail trap dashboard.

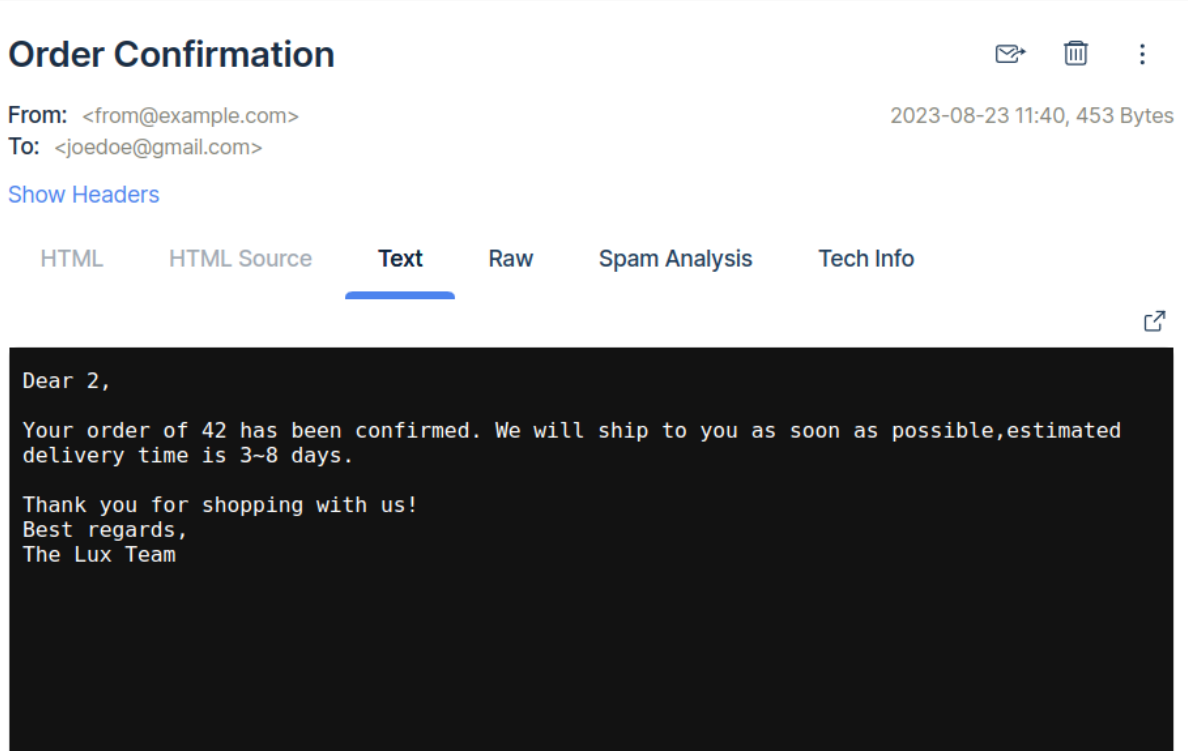
```
 EMAIL_BACKEND =
'django.core.mail.backends.console.EmailBackend'
```

Update the webhook endpoint to send an email to the user after payment is succesful

```
if event['type'] == 'checkout.session.completed':
    payment_intent = event['data']['object']


    user_id = payment_intent['client_reference_id']
    order_id = payment_intent['metadata']['order_id']

    order =  get_object_or_404(Order, id = order_id)
    order.payment_id = payment_intent['payment_intent']
    order.paid =True
    amount_total_dollars = payment_intent['amount_total'] /
100
    order.amount = amount_total_dollars
    order.save()


    user = User.objects.get(id =user_id)
    Cart.objects.filter(customer=user).delete()


    # TO DO : send email

    subject = 'Order Confirmation'
    message = f"Dear {user_id},\n\n" \
              f"Your order of {order_id} has been
confirmed. We will ship to you as soon as possible," \
              "estimated delivery time is 3~8 days.\n\n" \
              "Thank you for shopping with us!\n" \
              "Best regards,\n" \
              "The Lux Team"
    from_email = 'from@example.com'
    recipient_list = [payment_intent['customer_email']]
```

```
    send_mail(subject, message, from_email, recipient_list,
fail_silently=False)
```

If you login to your Mailtrap account, you should see the Order Confirmation email.



CHAPTER 5 Authentication

In Chapter 2 we created a Custom user which look like this:

```
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    pass
```

UserCreationForm

Django already has a built-in `UserCreationForm`, which inherits all the authentication features django provides.

The `UserCreationForm` is a built-in form provided by the Django authentication framework that simplifies creating a new user account. It inherits all the authentication features django provides. It also handles the creation of new users as well as validating data on the backend.

The UserCreationForm typically only includes the fields for basic user registration. I.e.:

- Username
- Email
- Password1
- Password2

However, by inheriting from the `UserCreationForm`, we can add additional fields.

Using the `UserCreationForm` will make it easy to set up user registration without implementing the form from scratch.

In the users app, create a forms.py file and make these imports.

```
from django.shortcuts import render,redirect
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import get_user_model
```

`get_user_model` is a utility function provided by the django.contrib.auth module that allows you to get the currently active user model.

Next, create a `CustomUserCreationForm` class which inherits from the `UserCreationForm`.

```
User = get_user_model()

class CustomUserCreationForm(UserCreationForm):
    class Meta:
        model = User
        fields = ['username',
                  'email',
                  'first_name',
                  'last_name'
                  ]

    def __init__(self, *args, **kwargs):
        super(CustomUserCreationForm, self).__init__(*args,
**kwargs)
        self.fields['first_name'].required = True
        self.fields['last_name'].required = True
```

By default, Django forms require a Meta class, which defines the model and the fields attributes. We have also added custom fields `first_name` and `last_name`; we then override the `CustomUserCreationForm` and set the first and last names as required fields. The user must enter their first and last names when creating a new account in our application.

Register users

We already have everything we need to start creating new users. In **users/views.py,** import the `CustomUserCreationForm` and the inbuilt django `login` function.

```
from .forms import CustomUserCreationForm
from django.contrib.auth import login
```

Next, create a `register` function that renders an instance of the `CustomUserCreationForm` if the method is GET and gets the POST data from the form if the request method is POST.

```
def register(request):
    if request.method=="POST":
        form = CustomUserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            print(user)
            login(request, user)
            return redirect('/')
    else:

        form = CustomUserCreationForm()
    context = {'form':form}
    return render(request, 'users/registration.html',
context)
```

In a GET request, we create an instance of `CustomUserCreationForm` and add it to the context dictionary. A context allows us to send data from a view to a template in django.

In the POST method, we initialize the form with the data submitted via a POST request. If the data is valid, i.e., if it passes the validation features provided by django, we save the new user to the database, log them in with `login(request, user)`, and then redirect them to the home page.

Create the **registration.html** page following this structure in the users app.

```
users/
└── templates/
    └── users/
        └── registration.html
```

In the **registration.html** page, render the form passed in the context dictionary.

```
{% extends 'base.html' %}
{% load crispy_forms_tags %}
{% block content %}

<div class="container">
    <div class="row justify-content-center mt-5 ">
        <div class="col-md-6">
              <form action="{% url 'register'%}"
method="post"   class="form">
                {% csrf_token %}
                {{ form|crispy }}
                <button type="submit" class="btn btn-
primary">Sign Up</button>
              </form>
          </div>
      </div>
  </div>
{% endblock %}
```

Create a **users/urls.py** file and map the view to a url

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

urlpatterns = [
    path('register', views.register, name = 'register'),


]
```

In the root **urls.py** file, include the **users. urls**

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

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

Run the development server and navigate to

http://127.0.0.1:8000/register

Username*

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Email address

First name*

Last name*

Password*

- Your password can’t be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can’t be a commonly used password.
- Your password can’t be entirely numeric.

Password confirmation*

Enter the same password as before, for verification.

Sign Up

Login Page

The next step is to build a login page. In **users/views.py,** add the code below.

```
from django.contrib.auth.views import LoginView
from django.contrib.auth.forms import AuthenticationForm
class UserLoginView(LoginView):
    template_name = 'users/login.html'
    form = AuthenticationForm
```

One of the advantages of django is that it provides many built-in classes, methods, and functions that make it easier to implement features quickly. The django documentation says this about the `AuthenticationForm` and the `LoginView`

AuthenticationForm - Base class for authenticating users. Extend this to get a form that accepts username/password logins

Here's what `LoginView` does:

- ***If called via `GET`, it displays a login form that POSTs to the same URL. More on this in a bit.***
- ***If called via `POST` with user submitted credentials, it tries to log the user in. If login is successful, the view redirects to the URL specified in `next`. If `next` isn't provided, it redirects to `settings.LOGIN_REDIRECT_URL` (which defaults to `/accounts/profile/`). If login isn't successful, it redisplays the login form.***

The `AuthenticationForm` accepts a `username` and a `password`. The built-in `LoginView` class handles a user's login process and creates an auth session.

Create the **login.html** page in the templates directory and render the `AuthenticationForm` instance.

```
users/
  └── templates/
     └── users/
        └── registration.html
        └── login.html
```

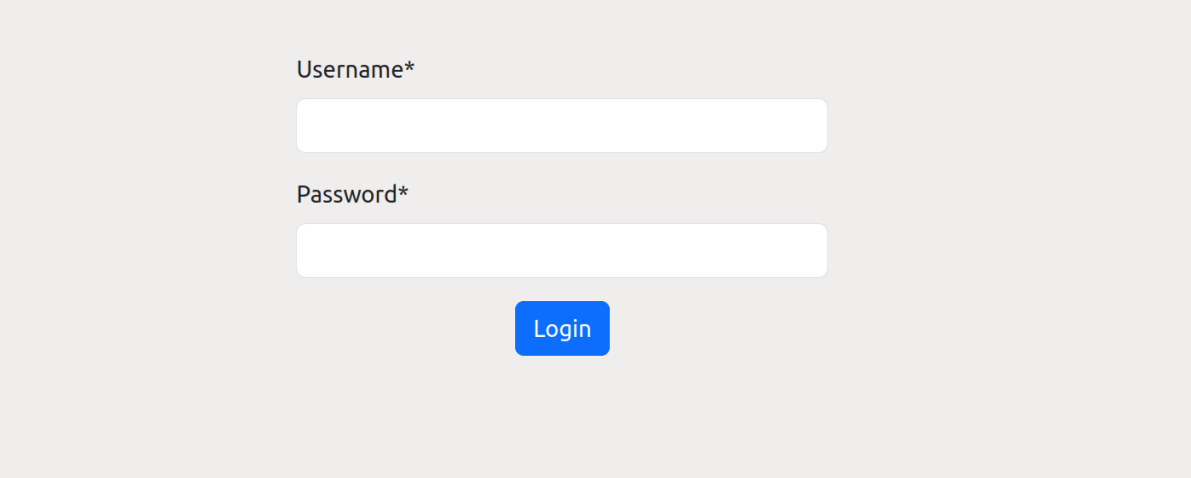
login.html

```
{% extends 'base.html' %}
{% load crispy_forms_tags %}
{% block content %}
<div class="container">
    <div class="row justify-content-center mt-5 ">
        <div class="col-md-6">
            <form action="{% url 'login' %}" method="post"
class="form">
                {% csrf_token %}
                {{ form|crispy }}

                <div class="form-group mt-3 text-center">
                    <button type="submit" class="btn btn-
primary">Login</button>
                </dv>
           </form>
        </div>
    </div>
</div>
{% endblock %}
```

Map the view to a url in **users/urls.py** file.

```
urlpatterns = [
    path('register', views.register, name = 'register'),
    path('login', views.UserLoginView.as_view(), name =
'login'), #new

]
```

Navigate to http://127.0.0.1:8000/login and you can see the login form.



After users log in, we must direct them to a particular page. In django, the redirect after login is defined in the settings.py file.

```
LOGIN_REDIRECT_URL ='/'
```

Logout Users

To log out a user and clear their session data, django provides the inbuilt `logout` function. Let's create the logout_user view function.

```
from django.contrib.auth import login,logout
def logout_user(request):
   logout(request)
   return redirect("login")
```

Map the view to a url.

```
urlpatterns = [
    path('register', views.register, name = 'register'),
    path('login', views.UserLoginView.as_view(), name =
'login'),
    path('logout', views.logout_user, name = 'logout'),
#new

]
```

In the **base.html** form, write some logic code that will show the logout button when the user is authenticated and if the user is not authenticated, the login/signup button will be shown.

```
<ul class="navbar-nav">
            <a class="nav-link" href="">
              <i class="fa fa-shopping-cart"></i>
              <span class="cart-count">Cart</span>
            </a>
```

```
            {% if user.is_authenticated %}

            <li class="nav-item">
              <a class="nav-link" href="#">Hello {{user}}
</a>
            </li>

            <li class="nav-item">
              <a class="nav-link" href="{% url
'logout'%}">Logout</a>
            </li>
            {% else %}
            <li class="nav-item">
               <a class="nav-link" href="{% url
'register'%}">Login/SignUp</a>
            </li>
            {% endif %}
  </ul>
```

Password Reset

The django authentication system streamlines the password reset process. It provides inbuilt views which perform the required reset password functions behind the scenes.

1. **PasswordResetView**: displays a password reset where a user can provide their email for password reset
2. **PasswordResetDoneView**: After a user enters their email, django displays the `PasswordResetView` that informs the user that a password reset email has been sent.

3. **PasswordResetConfirmView**: The email sent to the provided email contains a link with a unique token. The **PasswordResetConfirmView** handles the password reset confirmation by verifying the validity of the password reset token. The user is then able to enter a new password.
4. **PasswordResetCompleteView**: This view informs the user that the password rest has been successful.

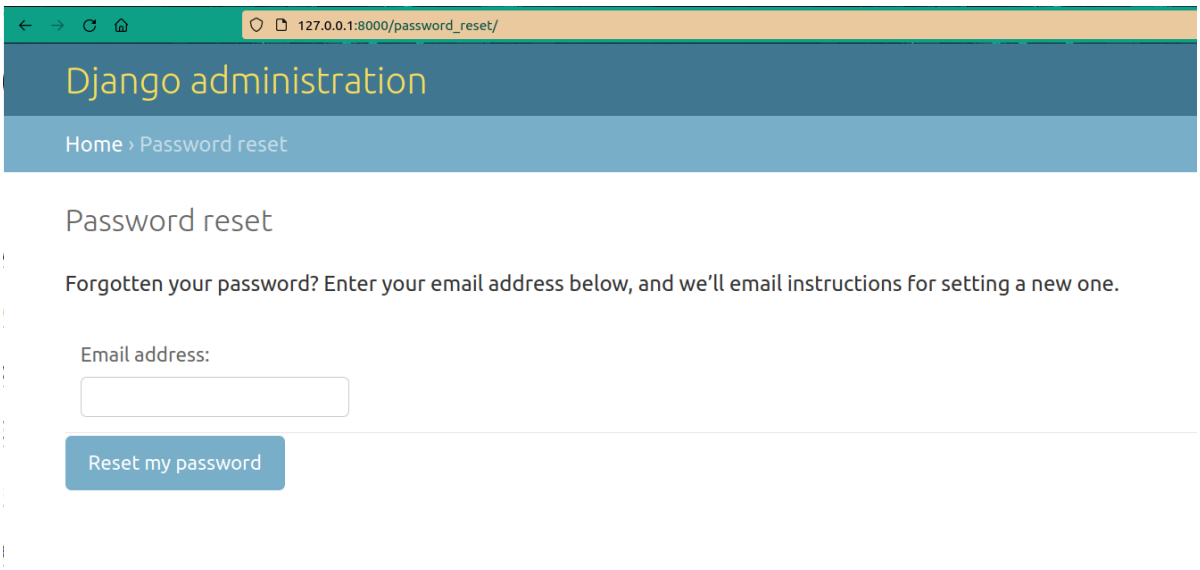
The inbuilt views handle the handles entire password reset process, which includes:

- Generating a unique token
- Sending the unique token to the user emails email
- validating the token
- Updating a new password

We need to add the following urls to the **users/urls.py** file to use the inbuilt password rest views.

```
from django.contrib.auth import views as auth_views
urlpatterns = [
    # other urls

    path('password_reset/',
auth_views.PasswordResetView.as_view(),
name='password_reset'),    path('password_reset/done/',
auth_views.PasswordResetDoneView.as_view(),
name='password_reset_done'),
path('reset/<uidb64>/<token>/',
auth_views.PasswordResetConfirmView.as_view(),
name='password_reset_confirm'), path('reset/done/',
auth_views.PasswordResetCompleteView.as_view(),
name='password_reset_complete')
]
```

If you navigate to one of the urls, for example, password_reset, you should see something like this:



By default, the reset password views use the django admin interface. But we dont want our users to use the admin interface to reset the password; therefore, we will customize the html pages by defining custom rest password html pages.

Let’s start by creating the following html pages.

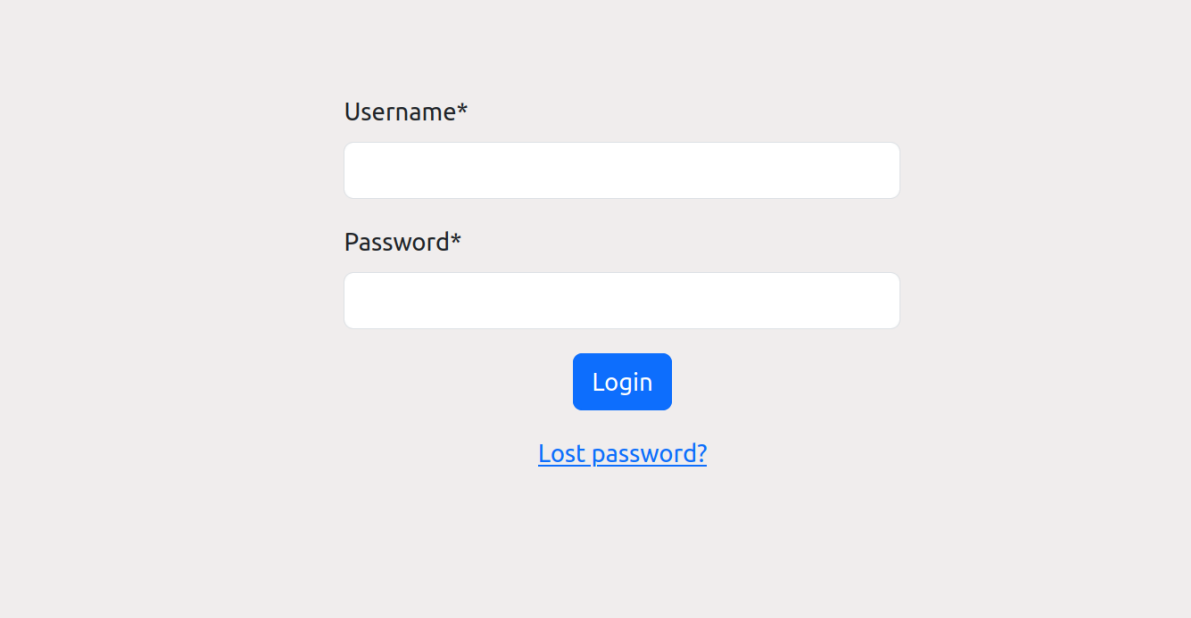
```
users/
└── templates/
    └── users/
        ├── password-reset-complete.html
        ├── password-reset-confirm.html
        ├── password-reset-done.html
        ├── password-reset.html
```

In the **users/urls.py file**, add each template to its corresponding url

```
urlpatterns = [
    # other urls
    path('password_reset/',
auth_views.PasswordResetView.as_view(template_name =
'users/password-reset.html'), name='password_reset'),
    path('password_reset/done/',
auth_views.PasswordResetDoneView.as_view(template_name =
'users/password-reset-done.html'),
name='password_reset_done'),
    path('reset/<uidb64>/<token>/',
auth_views.PasswordResetConfirmView.as_view(template_name =
'users/password-reset-confirm.html'),
name='password_reset_confirm'),
  path('reset/done/',
auth_views.PasswordResetCompleteView.as_view(template_name
= 'users/password-reset-complete.html'),
name='password_reset_complete'),

]
```

In the **login.html** page, add a button linking to the `reset-password` url

```
 <div class="form-group mt-3 text-center">
        <p><a href="{% url 'password_reset' %}">Lost
password?</a></p>
    </div>
```



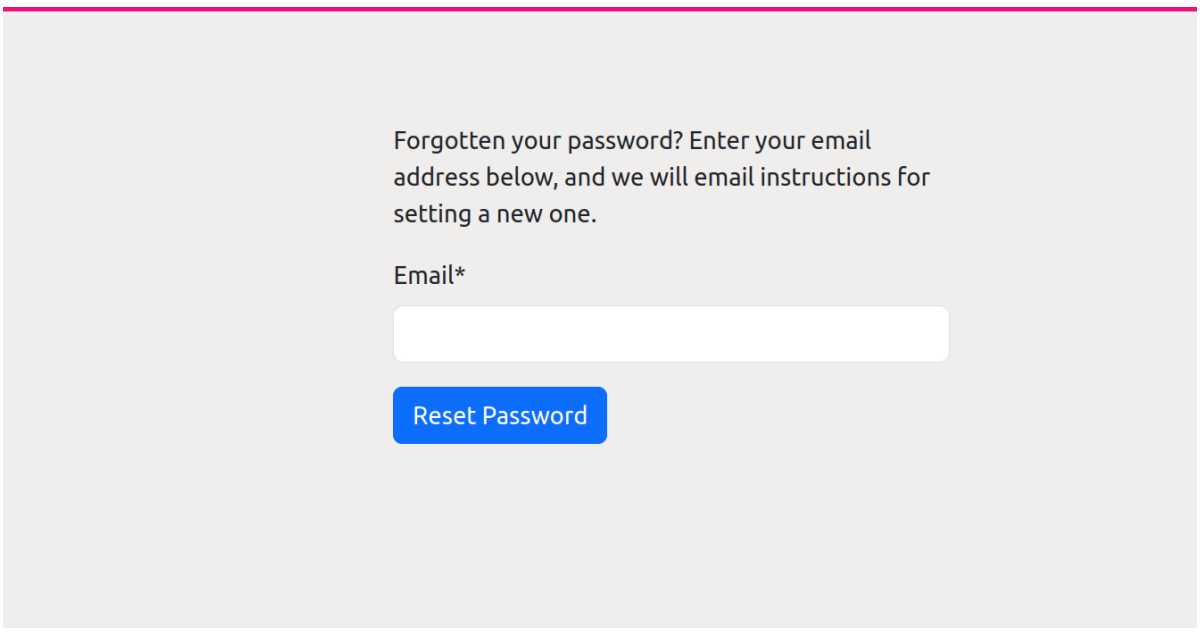
In the **password-reset.html** page, add the reset password form.

password-reset.html

```
{% extends 'base.html' %}
{% load crispy_forms_tags %}
{% block content %}
```

```
<div class="container">
    <div class="row justify-content-center mt-5 ">
        <div class="col-md-6">
            <p>Forgotten your password? Enter your email
address below, and we will email instructions for setting a
new one.</p>
              <form action="#" method="post" class="form">
                  {% csrf_token %}
                  {{ form|crispy }}

                  <button type="submit" class="btn btn-
primary">Reset Password</button>
              </form>
          </div>
      </div>
  </div>
{% endblock %}
```

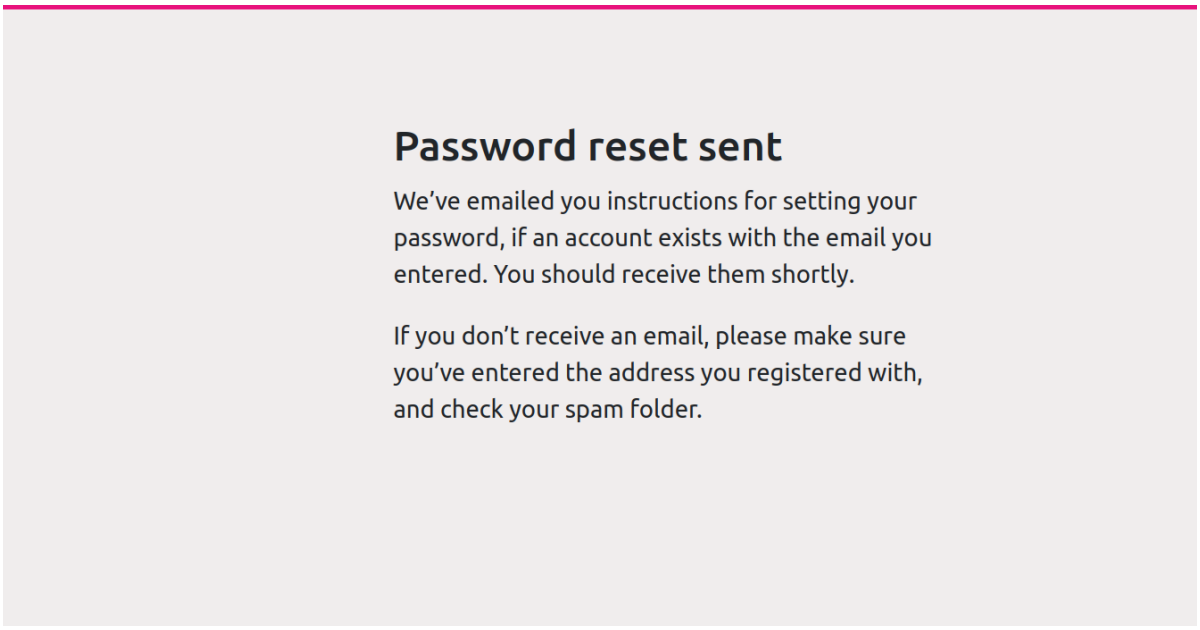
The password reset page looks like this:



After a user submits their password, they will be redirected to a page that informs them that the password reset email has been sent. In the **password-reset-done.html** page, add the contents below.

```
{% extends 'base.html' %}
{% load crispy_forms_tags %}
{% block content %}

<div class="container">
    <div class="row justify-content-center mt-5 ">
        <div class="col-md-6">
            <h3>Password reset sent</h3>
              <p>
We've emailed you instructions for setting your password,
if an account exists with the email you entered. You should
receive them shortly.</p>
<p>
    If you don't receive an email, please make sure you've
entered the address you registered with, and check your
spam folder.
</p>
          </div>
      </div>
  </div>
{% endblock %}
```

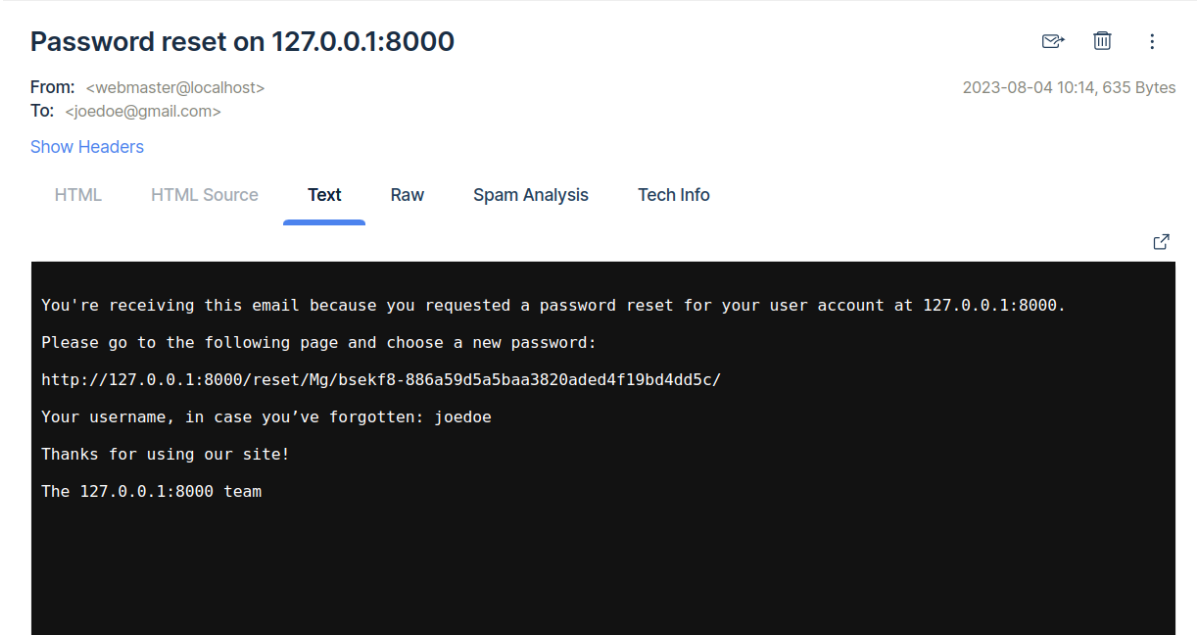
The password-rest done page looks like this:



The following will also happen behind the scenes:

- Django will first check if the user exists and they have a valid email address saved. If true, with the help of the email-sending feature, a unique token will be generated and sent to the user's email address.
- The email contents will have the link containing the unique token; when the user clicks on the link, they will be redirected to a page where they can reset their password.

Now, go to your MailTrap dashboard and see the reset password email.



The link http://127.0.0.1:8000/reset/Mg/bsekac-5728d03fd33855b1bc092f409d97dcab/ contains a one-time unique token that lets the user reset their password. Users will be redirected to the password-reset-confirm page when they click the link.

If you are using the console to print emails, you should see such a message in the terminal.

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 8bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: joedoe@gmail.com
Date: Fri, 04 Aug 2023 10:17:05 -0000
Message-ID: <169114422583.29917.9039416447120117653@vaati>

You're receiving this email because you requested a
password reset for your user account at 127.0.0.1:8000.

Please go to the following page and choose a new password:
```

```
http://127.0.0.1:8000/reset/Mg/bsekkh-
95169f5b09d6d12d156fc03cc19cbfd7/

Your username, in case you've forgotten: joedoe

Thanks for using our site!

The 127.0.0.1:8000 team
```

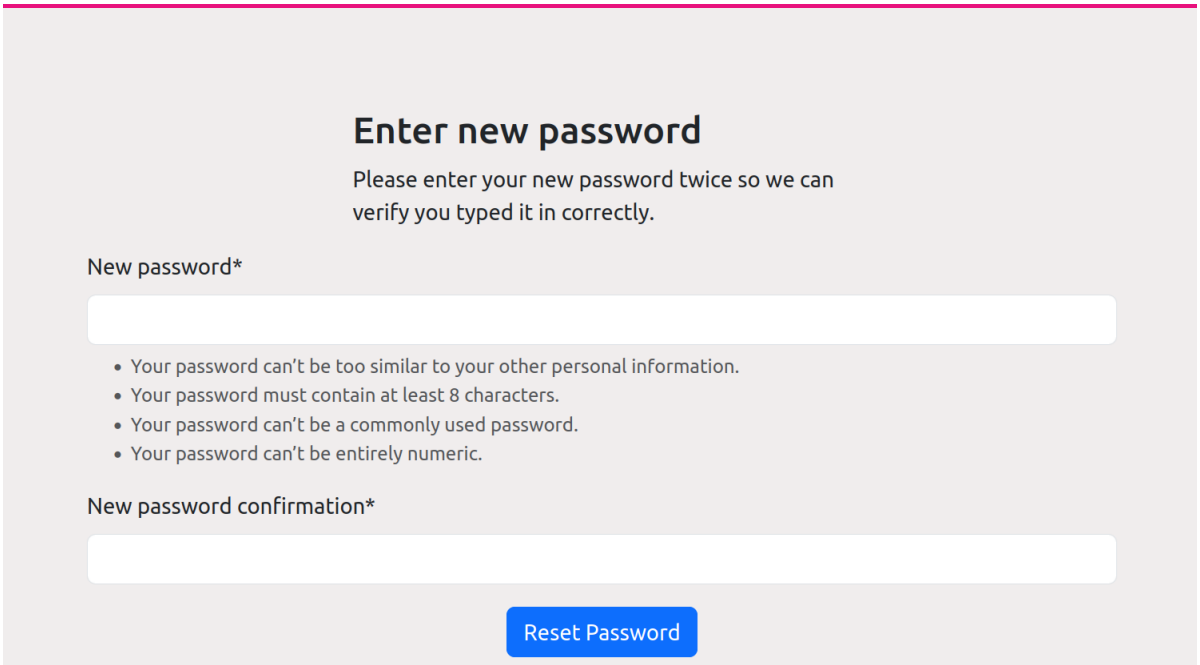
In the **password-reset-confirm.html**, add the password reset confirm form.

password-reset-confirm.html

```
{% extends 'base.html' %}
{% load crispy_forms_tags %}
{% block content %}

<div class="container">
    <div class="row justify-content-center mt-5 ">
        <div class="col-md-6">
            <h3>Enter new password</h3>
              <p>Please enter your new password twice so we
can verify you typed it in correctly.</p>
          </div>
          <form action="#" method="post" class="form">
            {% csrf_token %}
            {{ form|crispy }}

            <div class="form-group mt-3 text-center">
                <button type="submit" class="btn btn-
primary">Reset Password</button>
            </div>

        </form>
```

```
      </div>
  </div>
{% endblock %}
```

The form should look like this:



You will get an error belowif you attempt to use the link again.

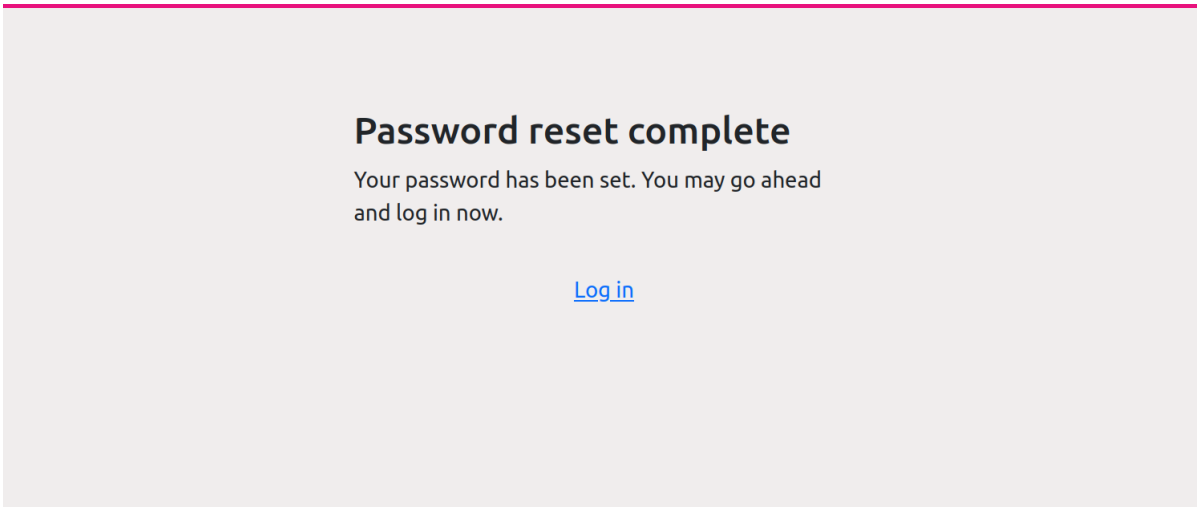
```
"The password reset link was invalid, possibly because it has already been used. Please request a new password reset"
```

After submitting the new password, the user will be redirected to the password-reset-complete page. Add the following contents to the **password-reset-complete.html** page.

password-reset-complete.html

```
{% extends 'base.html' %}
{% load crispy_forms_tags %}
{% block content %}

<div class="container">
    <div class="row justify-content-center mt-5 ">
        <div class="col-md-6">
            <h3>Password reset complete</h3>
              <p>Your password has been set. You may go
ahead and log in now.</p>
          </div>
      </div>
      <div class="form-group mt-3 text-center">
        <p><a href="{% url 'login' %}">Log in</a></p>
    </div>
  </div>
{% endblock %}
```

The page looks like this:



Orders page

We need customers to see their order history and also the status of their orders. In the store/models.py file, add a status field .

```
SHIPPING_STATUS_CHOICES = (

        ('Pending', 'Pending'),
        ('Shipped', 'Shipped'),
        ('Delivered', 'Delivered'),

    )
status = models.CharField(max_length=20,
            choices=SHIPPING_STATUS_CHOICES,
            default='Pending'
)
```

Run migrations to update the fields.

```
 python manage.py makemigrations
```

You should see this output.

```
Migrations for 'orders':
  orders/migrations/0004_order_status.py
    - Add field status to order
```

Run the migrate command.

python manage.py migrate

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, orders,
sessions, store, users
Running migrations:
  Applying orders.0004_order_status... OK
```

In the orders/views.py file, create an OrderListView class that returns all orders for the current customer.

```
from django.views.generic import ListView

class OrderListView(ListView):
    template_name = 'orders/orders.html'
    model = Order
    context_object_name = 'orders'

    def get_queryset(self):
        return Order.objects.filter(
               customer=self.request.user)
               .order_by('date_created')
```

The OrderListView class is a subclass of Django's ListView class. `Order.objects.filter(customer=self.request.user).order_by('date_created')` retrieves the list of orders for the currently logged-in user. The orders are ordered by date_created in ascending order.

Create the `order_list.html` page and render the orders in a Bootstrap table.

```
{% extends 'base.html'%}
{% block content%}
{% if orders %}

<div class="container mt-4">
  <div class="row">
     <h4 class="text-center">Order History</h4>
  </div>
</div>
<div class="container mt-4">

    <div class="row">
      <div class="col-12">
        <div class="table-responsive">
```

```
          <table class="table table-bordered table-
striped">
            <thead class="thead-dark">
              <tr>
                <th>Order No.</th>
                <th>Order Date</th>
                <th>Status</th>
                <th>Paid</th>
                <th>Details </th>

              </tr>
            </thead>
            <tbody>
              {% for order in orders %}
                <tr>
                  <td># {{ order.id }}</td>
                  <td>{{ order.date_created }}</td>
                  <td>{{ order.status }}</td>
                  <td>$ {{ order.amount}}</td>
                  <td>
                    <a href="#">View Details</a>
                  </td>

                </tr>
              {% endfor %}
            </tbody>
          </table>
        </div>
      </div>
    </div>
</div>


{% else%}
<div class="container mt-5 text-center">
  <div class="row">
     <div class="col-md-6 offset-md-3">
        <div class="card">
           <div class="card-body">
              <p class="card-text">No orders</p>
```

```
              <a href="/" class="btn btn-primary">Continue
Shopping</a>
           </div>
        </div>
     </div>
  </div>
</div>

{% endif%}
{% endblock%}
```

In the order_list.html page, we first check the orders list; if the list is not empty, we loop over the list and display the following information.

- The order number
- The order status
- The total amount paid
- A link for more details about the order

If the list is empty, we display the message “No orders” and a link to the products page.

Map the view to a url. In order/urls.py update it as follows.

```
urlpatterns = [



path('checkout',views.CreateCheckoutSessionView.as_view(),
name='checkout' ),
    path('success',views.success_page, name='success' ),
    path('cancel',views.cancelled_page, name='cancel' ),
    path('webhooks',views.stripe_webhook, name='webhooks'
),
    path('orders',views.OrderListView.as_view(),
name='orders' ), #new

]
```

If you navigate to the http://127.0.0.1:8000/orders/ page, you should see an history of order displayed.

Pagination

We need to add pagination links at the bottom to make the orders page more user-friendly.
Django has a built-in paginator class that allows us to paginate a list of objects
The Django paginator class takes a list of objects and the number of items to be paginated on each page; for example, if we need to paginate a list of orders:

```
from django.core.paginator import Paginator
objects = Order.objects.all()
paginator = Paginator(objects, 10)
page_number = request.GET.get(“page”)
page_obj = paginator.get_page(page_number)
```

Here we are telling django to fetch ten orders on each page.
`page_number = request.GET.get(“page”)`: will get the page_number clicked by the user on the template.

For example, page 1 will be:

```
>>> page_obj = paginator.get_page(1)
>>> page_obj
<Page 1 of 1>
>>>
```

`page_obj` stores the specific page the user requests. It also manages pagination details, such as the total number of pages.
Then in the templates, you would pass the paginator class like this:

```
<div class="pagination">
  <span class="step-links">
    <span class="current">
      Page {{ page_obj.number }} of {{
page_obj.paginator.num_pages }}.
  </span>
</div>
```

Django's `ListView` generic class provides an easier way to paginate the list of objects. The paginate_by attribute limits the number of items per page to a specified number and adds the `paginator` and the `page_obj` to the context.
In the `OrderListView` class, add the paginate_by attribute.

```
class OrderListView(ListView):
    template_name = 'orders/order_list.html'
    model = Order
    context_object_name = 'orders'
    paginate_by =20

    def get_queryset(self):
        return Order.objects.filter(
           customer=self.request.user).
           order_by('date_created')
```

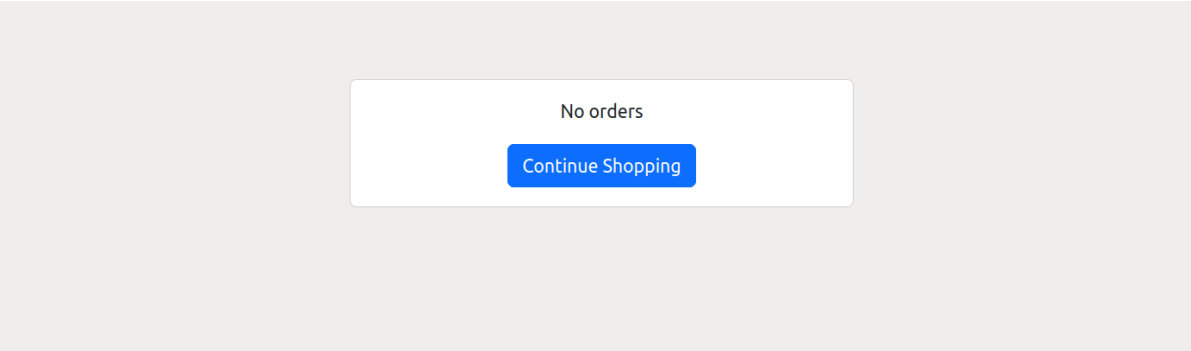
In the order_list.html page, update to include the pagination page and links. The pagination will be after the table container. The order_list.html page should now look like this:

```
<div class="container mt-4">
  <div class="row">
    <div class="col-12">
      <div class="table-responsive">
```

```
      <!--  orders table -->
        </table>
      </div>
    </div>
  </div>
</div>


<div class="pagination">
<span class="step-links">
    {% if page_obj.has_previous %}
        <a href="?page=1">&laquo; first</a>
        <a href="?page={{ page_obj.previous_page_number
}}">previous</a>
    {% endif %}

    <span class="current">
        Page {{ page_obj.number }} of {{
page_obj.paginator.num_pages }}.
    </span>

    {% if page_obj.has_next %}
        <a href="?page={{ page_obj.next_page_number
}}">next</a>
        <a href="?page={{ page_obj.paginator.num_pages
}}">last &raquo;</a>
    {% endif %}
</span>
</div>
```

Refresh the orders page, and you should see the pagination is now active.

If a customer has no order history, they will see this page.



Order Items Page

Each order item should furher link to a page containing al the order items . In orders/views.py add another class to display more details about an order.

```
class OrderItemView(ListView):
    template_name = 'orders/order_detail.html'
    context_object_name = 'order_items_list'

    def get_queryset(self):
        order_id = self.kwargs['id']
        order = get_object_or_404(Order, id=order_id)
        return order.order_items.all()
```

The OrderItemView retrieves the list of order items for the order id passed in the request. The queryset is then added to the context.

Create the order_detail.html page in the templates folder

```
orders/
└── templates/
    └── orders/
        ├── checkout.html
        ├── order_detail.html
        ├── order_list.html
```

Add the following code to order_detail.html page.

```
{% extends 'base.html'%}
{% block content%}

<div class="container mt-4">
  <div class="row">
     <h4 class="text-center">Order Summary</h4>
  </div>
</div>
<div class="container">
```

```
  {% for order_item in order_items_list %}
    <div class="row">
      <div class="col-12">
        <div class="card">
          <div class="card-body">
            <!-- Order item content goes here -->
            <img src="{{ order_item.product.image.url }}"
alt="Winter Landscape" width="60" class="img-fluid rounded"
style="max-width: 100%;" />
            <h5 class="card-title">{{
order_item.product.name }}</h5>
            <p class="card-text">Order No. #{{
order_item.order.id }}</p>
            <p class="card-text">Qty : {{
order_item.quantity }}</p>
            <p class="card-text">${{ order_item.subtotal }}
</p>
             <p class="card-text">Updated on: {{
order_item.order.date_updated }}</p>
             <p class="card-text">Status: {{
order_item.order.status }}</p>
          </div>
        </div>
      </div>
    </div>
  {% endfor %}
</div>
{% endblock%}
```

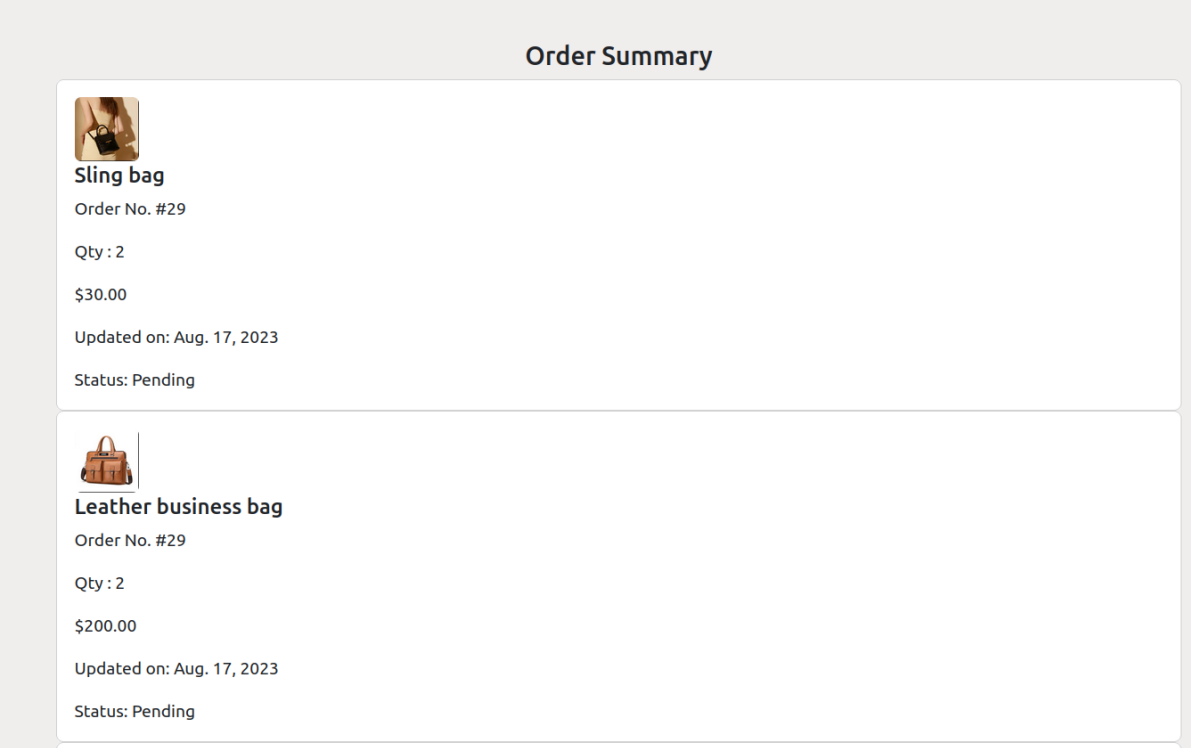
Map the view to a url.

```
path('orders/<int:id>/', views.OrderItemView.as_view(),
name='order-detail'),
```

Update the url in the orders_list.html page on the Details row.

```
<tbody>
  {% for order in orders %}
    <tr>
      <td># {{ order.id }}</td>
      <td>{{ order.date_created }}</td>
      <td>{{ order.status }}</td>
      <td>$ {{ order.amount}}</td>
      <td>
        <a href="{% url 'order-detail' order.id %}">View
Details</a>
      </td>

    </tr>
  {% endfor %}
</tbody>
```

Now when you click the View Details of an order, you will see a page like this:



Add Products to Wishlist

A wishlist functionality that allows users to add products they love to their wishlist. A wishlist creates a personalized experience for users.

Let’s create a Wishlist model in orders/models.py

```
class WishList(models.Model):
    customer = models.ForeignKey(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE,related_name='customer_wishlist')
    product = models.ForeignKey(Product, on_delete=
models.CASCADE,
                related_name='product_wishlist')

    def __str__(self):
        return str(self.product)
```

A wishlist model contain the customer_id and the product_id

Run migrations

```
python manage.py makemigrations
```

When you run the makemigrations command, you should see this output.

```
Migrations for 'orders':
  orders/migrations/0006_wishlist.py
    - Create model WishList
```

```
python manage.py migrate
```

This is the output of the migrate command.

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, orders,
sessions, store, users
Running migrations:
  Applying orders.0006_wishlist... OK
```

Create a function for adding products to a wish list. In orders/views.py , add the code below

```
from .models import Cart,WishList
def add_wishlist (request):
    """Add a product to the wishlist model"""

    if request.method =='POST':
        product_id = request.POST.get('id')
        wishlist = WishList.objects.filter(
             customer =
request.user,product_id=product_id).exists()
        if not wishlist:
            wish = WishList(
                 customer =
request.user,product_id=product_id
               )
            wish.save()
```

```
    return redirect('/')
```

In the add_wishlist function above, we first import the Wishlist model. If the request. method is POST; we get the product's id with `product_id =request.POST.get('id')`.

`wishlist = WishList.objects.filter(customer = request. user,product_id=product_id).exists()`, checks if the product exists in the WishList model; if it doesn't exist, we create a new wishlist object, save it to the database and redirect the user to the homepage.

Map the view to a url in the orders/urls.py file,

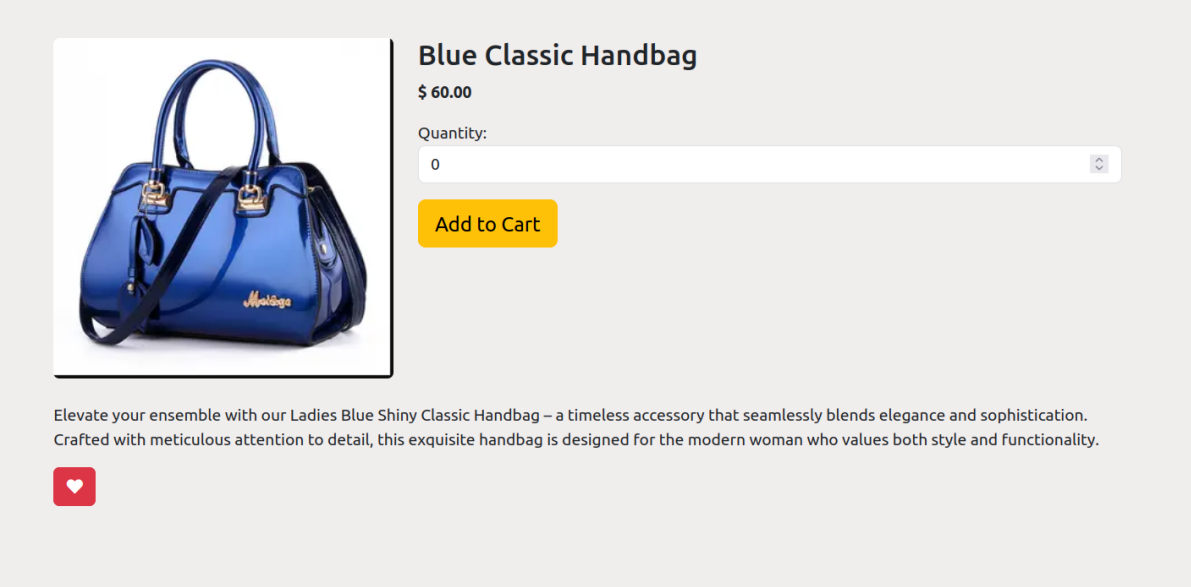
```
 path('add_wish',views.add_wishlist, name='add-wish' ),
#new
```

In the product detail.html page, add a form that lets the user add the product to the wishlist model. Open product_detail.html page and add the form after the description.

```
<!-- product_detail.html -->
<div class="container mt-4">
   <div class="row">

      <p>{{ product.description }}</p>
      <form action="{% url 'add-wish' %}" method="post">
         {% csrf_token %}
         <input type="text" name="id" value="{{ product.id
}}" class="form-control d-none">
         <button class="btn btn-danger" type="submit">
           <i class="fas fa-heart"></i>
```

```
          </button>
        </form>

    </div>
</div>
```

We are using a heart icon from the font -awesome to display a red heart which, when clicked, will add the product to a wishlist if it doesn't already exist. The product detail page now looks like this:



Display Wishlist

We also need to create a wishlist page that displays all the products in a wish list for the currently authenticated user. In orders/views.py, add the wishlist function:

```
def wishlist (request):
    """Display all products in the wishlist for the logged
in user"""

    wishlist = WishList.objects.filter(customer =
request.user)
    context = {'wishlist':wishlist}
    return render(request,'orders/wish_list.html', context)
```

Create the wishlist.html page in the templates folder.

```
orders/
└── templates/
    └── orders/
        ├── checkout.html
        ├── order_detail.html
        ├── order_list.html
        └── wish_list.html
```

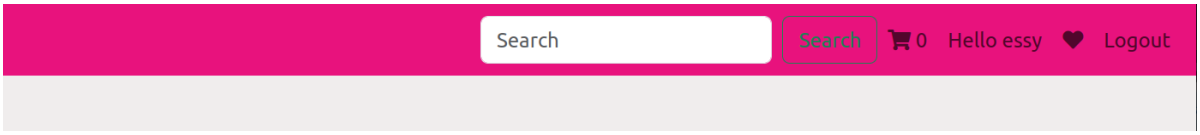
Map the view to a url

```
 path('wishlist',views.wishlist, name='wishlist' ), #new
```

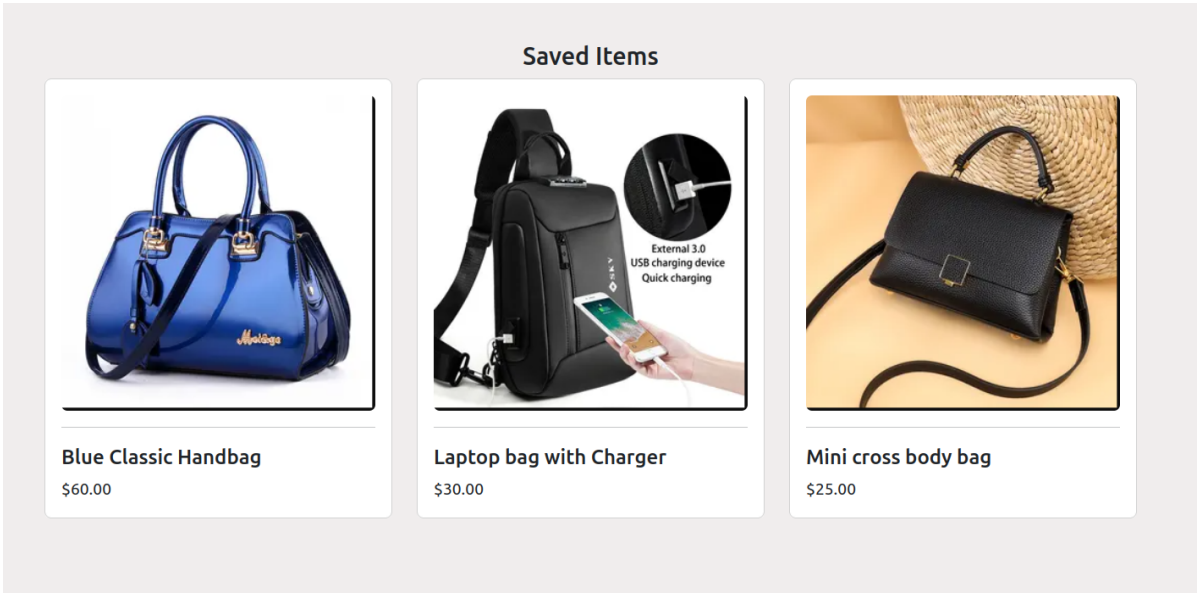
In the base.html file, add a heart icon wrapped in anchor tags that links to the wishlist page.

```
<a href="{% url 'wishlist' %}" class="nav-link">
<i class="fas fa-heart"></i>
</a>
```

The navbar now looks like this:



And the wishlist page looks like this:



CHAPTER 6: DEPLOYMENT

You have finished your django application; now it's time to make your application available to the public. There are various hosting providers which make it easy. The most popular include:

- Digital Ocean
- Heroku
- Python anywhere
- Railway.
- AWS etc

We will use Railway to deploy our django application. Railway provides a free tier that we can use to deploy projects without incurring any additional costs.

Deployment Checklist

- The Django documentation provides a checklist that provides a guideline for deploying your Django application. You can issue the command to check what is required. Let's do that. Inside your Django application directory, issue the following command,

```
python manage.py check --deploy
```

You should get a reposnse like this:

```
System check identified some issues:

WARNINGS:
?: (security.W004) You have not set a value for the
SECURE_HSTS_SECONDS setting. If your entire site is served
only over SSL, you may want to consider setting a value and
enabling HTTP Strict Transport Security. Be sure to read
the documentation first; enabling HSTS carelessly can cause
serious, irreversible problems.
?: (security.W008) Your SECURE_SSL_REDIRECT setting is not
set to True. Unless your site should be available over both
SSL and non-SSL connections, you may want to either set
this setting True or configure a load balancer or reverse-
proxy server to redirect all connections to HTTPS.
?: (security.W009) Your SECRET_KEY has less than 50
characters, less than 5 unique characters, or it's prefixed
with 'django-insecure-' indicating that it was generated
automatically by Django. Please generate a long and random
value, otherwise many of Django's security-critical
features will be vulnerable to attack.
?: (security.W012) SESSION_COOKIE_SECURE is not set to
True. Using a secure-only session cookie makes it more
difficult for network traffic sniffers to hijack user
sessions.
?: (security.W016) You have
'django.middleware.csrf.CsrfViewMiddleware' in your
MIDDLEWARE, but you have not set CSRF_COOKIE_SECURE to
True. Using a secure-only CSRF cookie makes it more
difficult for network traffic sniffers to steal the CSRF
token.
?: (security.W018) You should not have DEBUG set to True in
deployment.
?: (security.W020) ALLOWED_HOSTS must not be empty in
deployment.
```

Open the settings.py file and set DEBUG to False

```
DEBUG = False
```

Environment Variables

Environment variables are values that are usually set in the OS environment and can be accessed by different applications.
Environment variables allow programmers to store sensitive data, such as API keys.
To set environment variables depends on your operating system. In MacOS /Linux, environment variables are set like this:

```
export SECRET_KEY="your_secret_key_value"
```

Then in your application, you access the SECRET_KEY like this:

```
import os
SECRET_KEY = os.environ.get("SECRET_KEY")
```

Since We have other private keys, such as the Stripe API_KEYS and Mailtrap Credentials. Update the settings.py as follows:

```
EMAIL_HOST_USER = os.environ.get("EMAIL_HOST_USER")
EMAIL_HOST_PASSWORD = os.environ.get("EMAIL_HOST_PASSWORD")
STRIPE_PUBLISHABLE_KEY =os.environ.get(
                        "STRIPE_PUBLISHABLE_KEY")
STRIPE_SECRET_KEY = os.environ.get("STRIPE_SECRET_KEY")
SECRET_KEY = os.environ["SECRET_KEY"]
```

Static Files in Production

In a development environment, Static files are automatically served by the development server. In production, Django serves static files from one directory; first, we must define a STATIC_ROOT setting in the settings.py file.

```
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

Then, run the collectstatic command to copy all the files to the STATIC_ROOT directory.
Run the command.

```
python manage.py collectstatic
```

If all goes well, you should get an output like this:

```
125 static files copied to '/home/Desktop/lux/staticfiles'.
```

Requirements

Your application should also have a requirements.txt file; Issue the pip freeze command to collect your project dependencies in a requirements.txt file.

```
pip freeze > requirements.txt
```

Your requirements.txt file now looks like this:

```
asgiref==3.7.2
certifi==2023.7.22
charset-normalizer==3.2.0
crispy-bootstrap5==0.7
Django==4.2.4
django-crispy-forms==2.0
idna==3.4
Pillow==10.0.0
requests==2.31.0
sqlparse==0.4.4
stripe==5.5.0
typing_extensions==4.7.1
urllib3==2.0.4
```

Railway Production Configurations

To deploy your Django application to Railway, we must provide a Procfile. A Procfile is a file that specifies the commands to be executed when an application is deployed. Create a Procfile in the root directory and add the following contents.

```
web: python manage.py migrate && python manage.py
collectstatic --no-input && gunicorn LUX.wsgi
```

web implies a web service that will handle HTTP traffic. The rest of the commands will do the following

- python manage.py migrate: The migrate command will update our database tables
- python manage.py collectstatic will collect static files into the staticfiles directory from which they will be served
- gunicorn LUX.wsgi : This command will start a gunicorn server and serve your django application. LUX is the name of the django application

We also need to install gunicorn and include it in our requirements.txt file. Install gunicorn with pip.

```
pip install gunicorn
```

Issue the pip freeze command to add gunicorn to the requirements.txt file.

```
pip freeze > requirements.txt
```

Your requirements.txt file should now look like this:

```
asgiref==3.7.2
certifi==2023.7.22
charset-normalizer==3.2.0
crispy-bootstrap5==0.7
Django==4.2.4
django-cors-headers==4.2.0
django-crispy-forms==2.0
gunicorn==21.2.0
idna==3.4
packaging==23.1
Pillow==10.0.0
requests==2.31.0
sqlparse==0.4.4
stripe==5.5.0
typing_extensions==4.7.1
urllib3==2.0.4
```

Next, create a runtime.txt file and add the Python version you are using in your django project.

```
python -3.11
```

Allowed Hosts

Open the settings.py file and add the following:

```
ALLOWED_HOSTS = ['*']
```

The ALLOWED_HOSTS setting means that we are allowing all hosts to send HTTP requests to our django application. If you were using a domain name, you would set allowed hosts to the domain name as shown below.

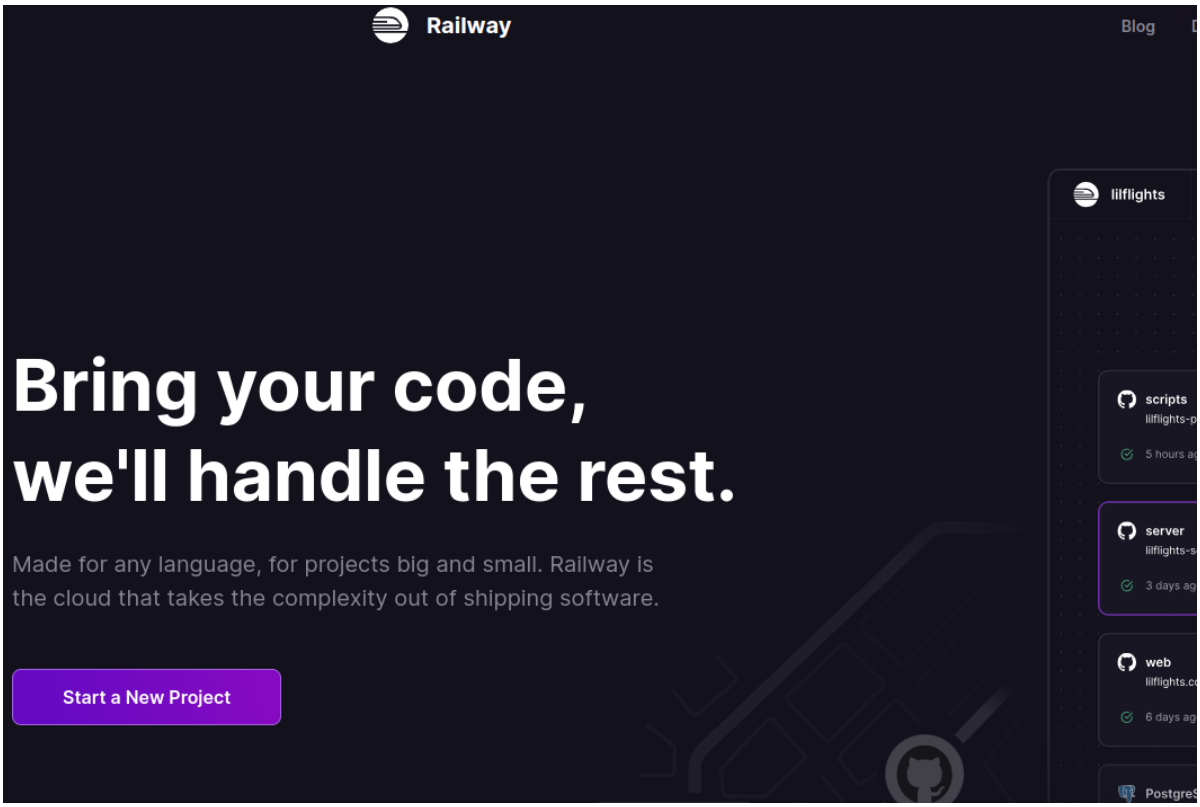
```
ALLOWED_HOSTS = ['example.com', 'www.example.com', 'ip-address]
```

We will update this later once we generate a domain from Railway. Your django application is now configured and ready for deployment on Railway.

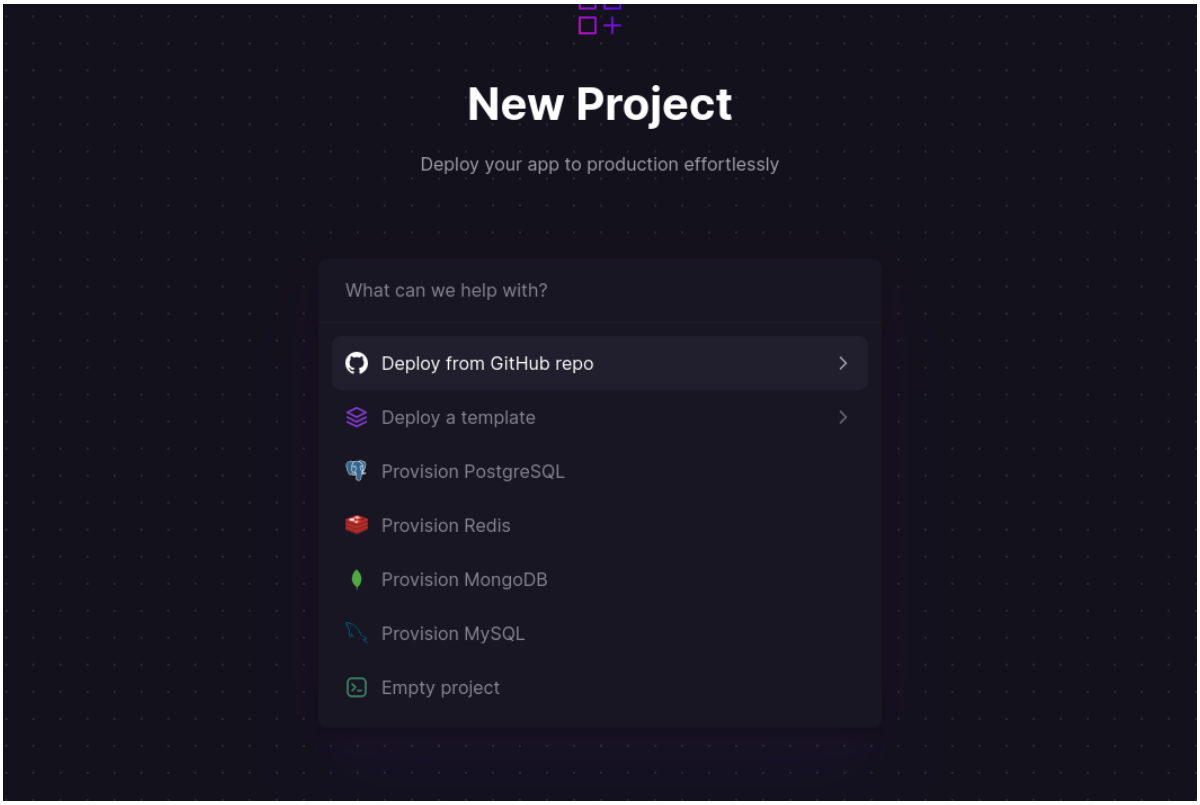
Create a Github Repo and a Railway Account

Railway allows you to automatically deploy changes to your GitHub repository if you have integrated the repository with Railway. If you dont already have a GitHub account, create one, create a new repository, and push your django project to Git Hub.

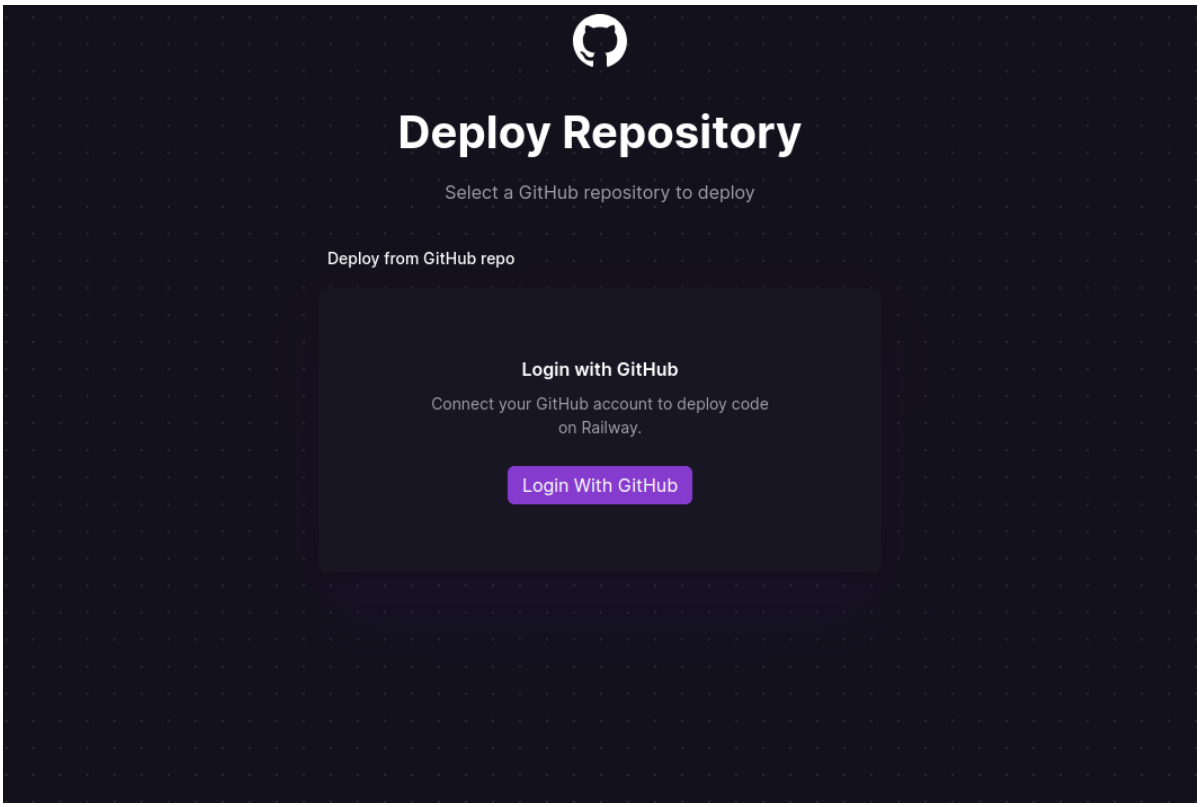
Next, go to the Railway site and start a New project.



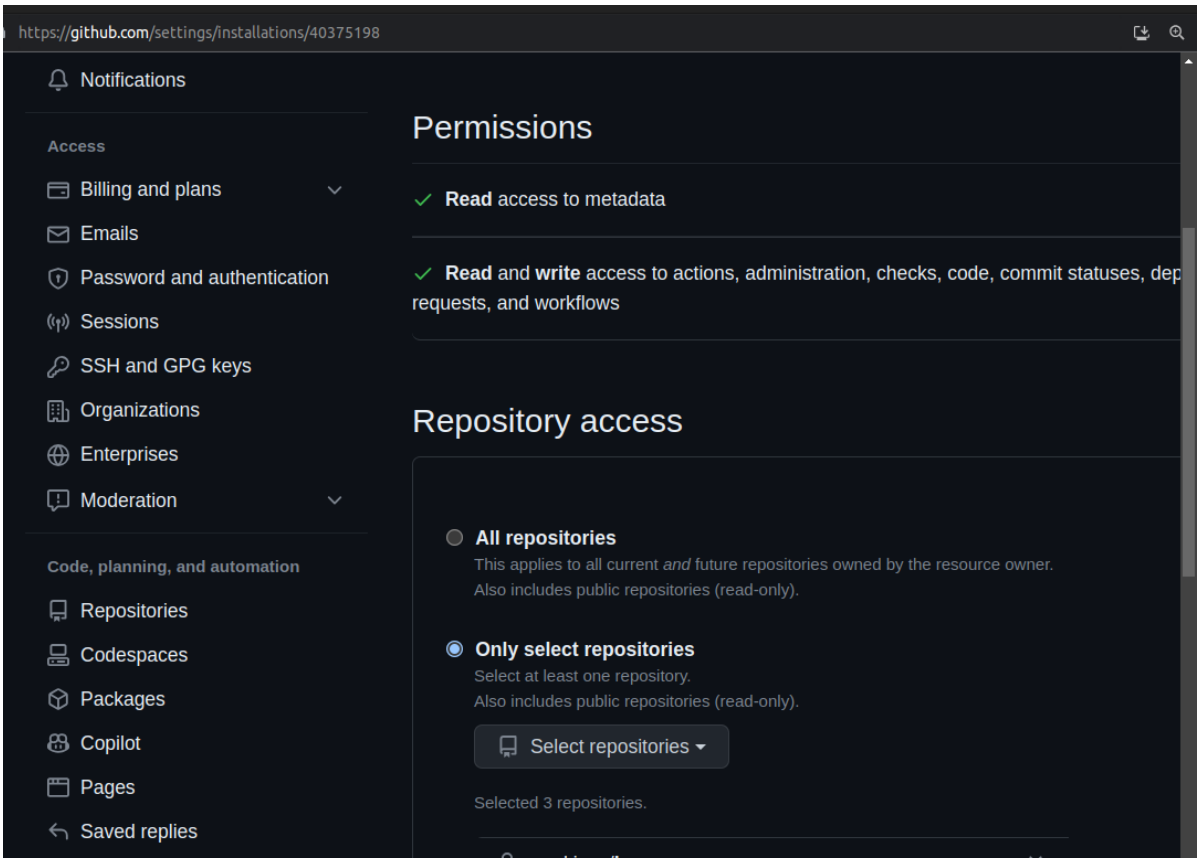
Click Start a New Project, and you will be redirected to this page.



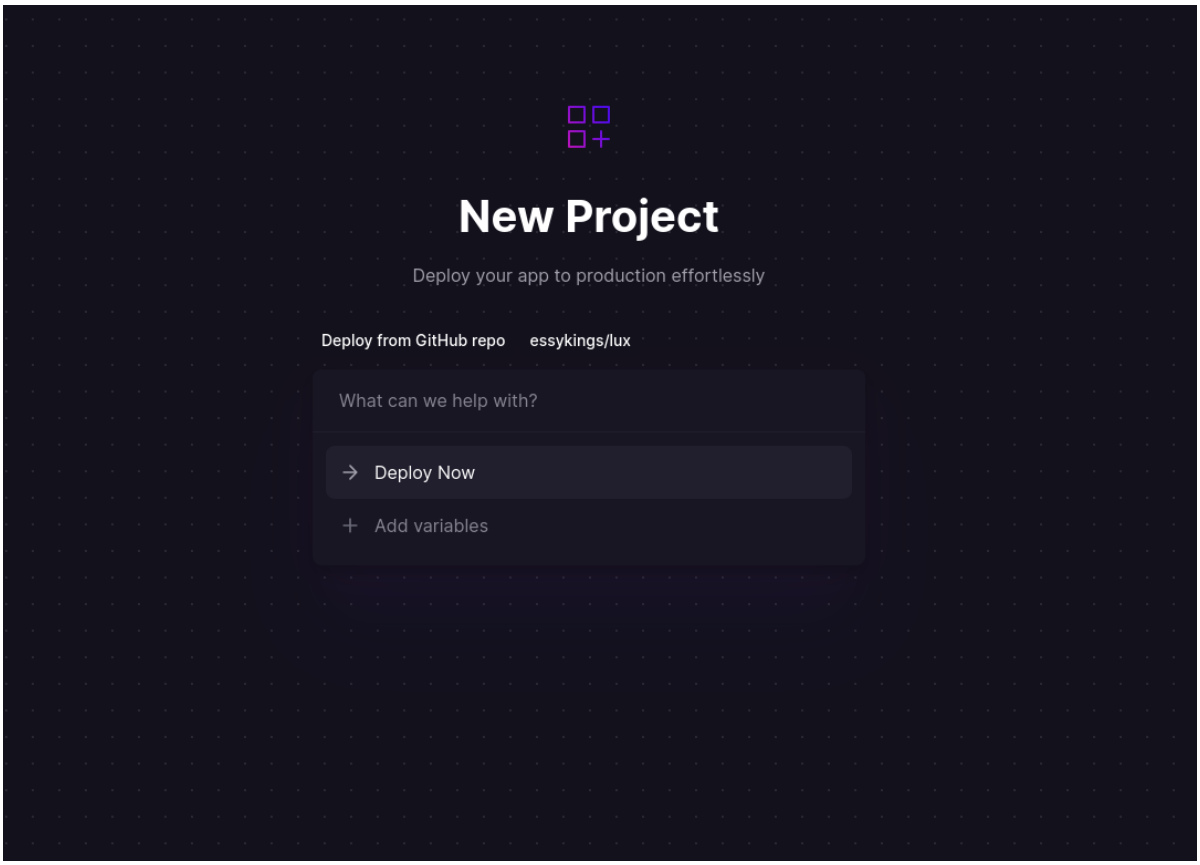
Select Deploy from the Github repo, and you will be redirected



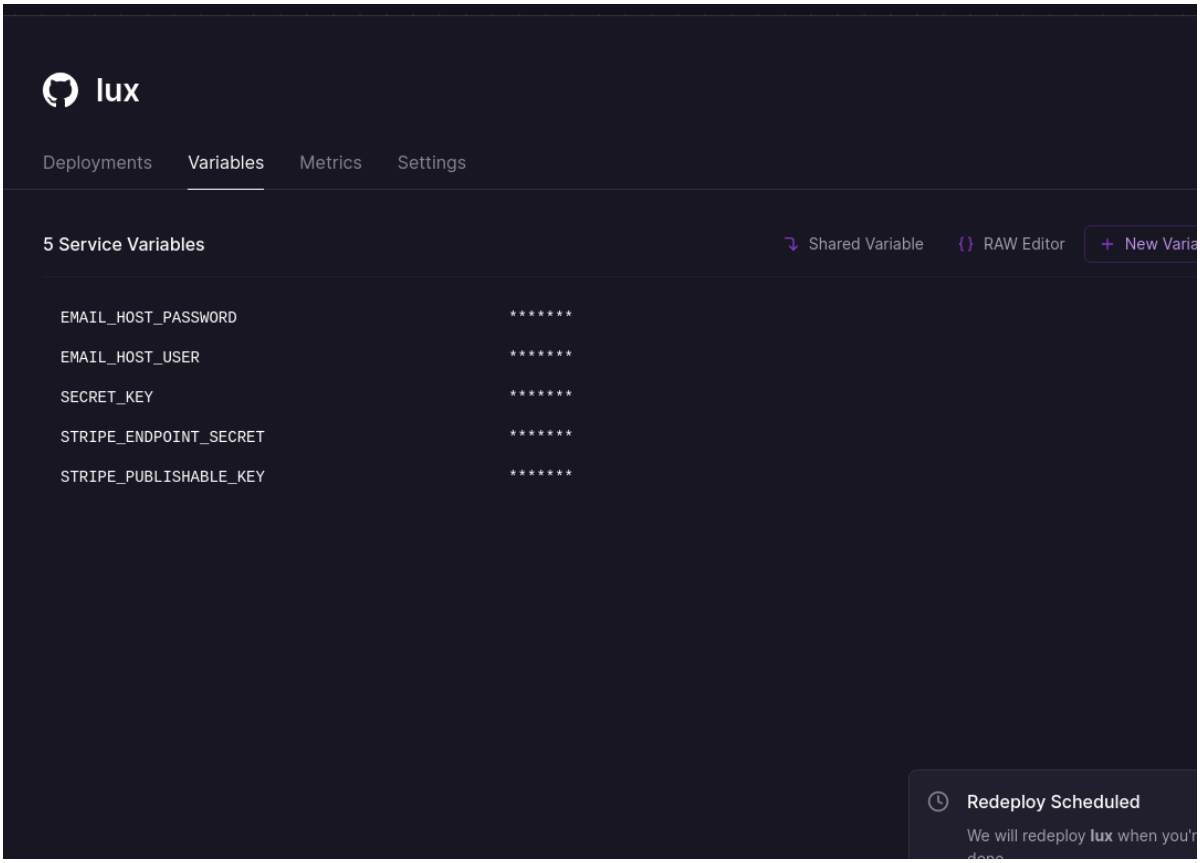
You can give Railway permission to all repositories or select a Specific Repository.



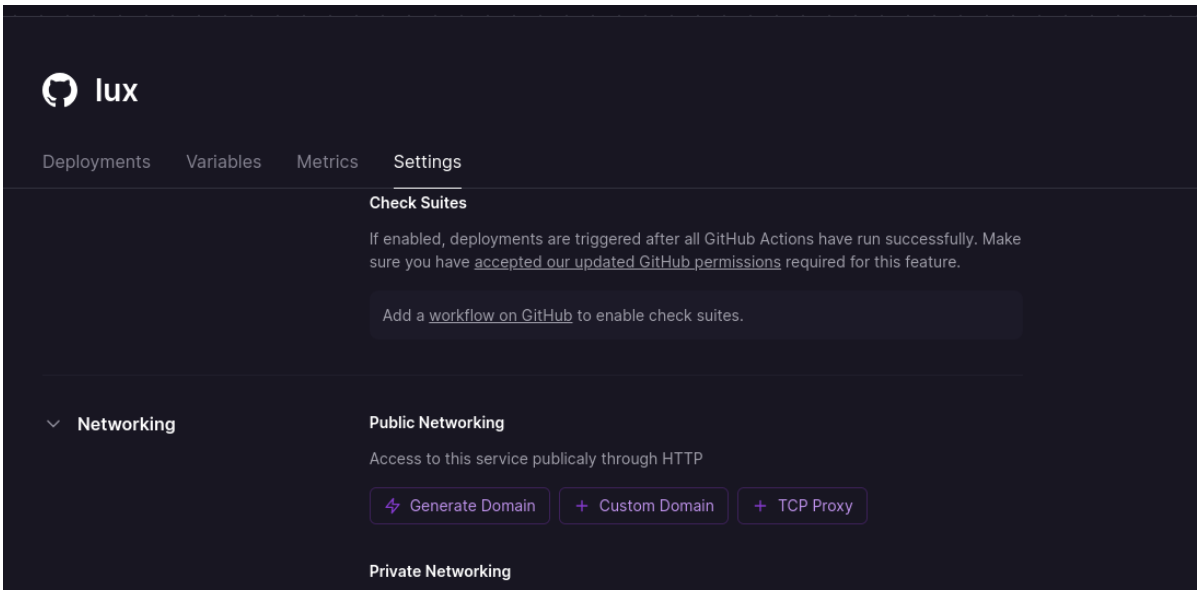
Once you select a Repository, you will be directed to this page.



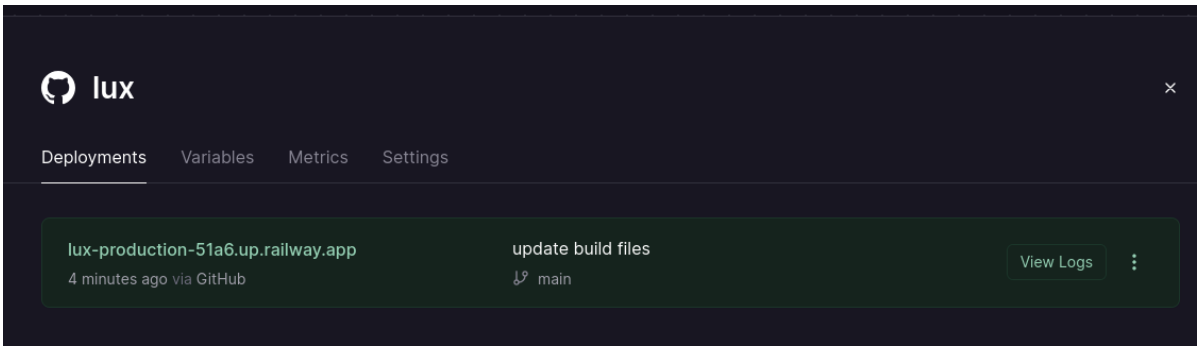
Before we hit deploy, let's add our environment variables.



Once you finish adding the environment variables, Go to settings and generate a domain.



Railway will be deploying in the background at this point, and you should see something like this.



Your Django application is now running at https://lux-production-51a6.up.railway.app/ . Now go back to your settings.py file and update ALLOWED_HOSTS to reflect the new domain.

```
ALLOWED_HOSTS = ['lux-production-51a6.up.railway.app/']
```

Additional resources

Django Docs

Simple is Better than Complex

Just Django

Adam Blog

Real Python

Testdriven.io

Learn Django

Conclusion

You can also find me on medium, where i release Django Tutorials every week. Your Feedback is very important, and if you notice any errors, you can raise an issue via Github.