E-commerce project

Virtual environment

We'll be using poetry to manage our virtual environment. If you haven't installed it yet, please follow the instructions for your OS here.

Creating the environment

On a console, browse to a folder of your choice and run (a new folder will automatically be created):

```
poetry new ecommerce-app
```

Enter the newly created folder:

```
cd ecommerce-app
```

Create/enter the virtual environment:

```
poetry shell
```

Installing packages

We will need some packages for our project, so let's install them in our environment. Remember that these will only be available inside the environment.

Make sure that you're inside the poetry shell, and run:

```
poetry add django flake8 black
```

Creating the django project

We'll use django as our web platform.

First we need to delete the existing `ecommerce-app` folder that poetry created. On linux/mac run:

```
rm -r ecommerce_app
```

On windows:

```
rmdir /s ecommerce_app
```

Let's create the project. Run these:

```
django-admin startproject ecommerce
```

```
cd ecommerce
```

Creating the app of our store

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

Add app to `settings.py`

Edit the `ecommerce-app/ecommerce/ecommerce/settings.py` file, and add the following to the INSTALLED_APPS list:

```
'store.apps.StoreConfig',
```

You should end up with this:

```
# Application definition

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

First run

Test if everything is good. Run `python manage.py runserver` on the terminal and open 127.0.0.1:8000 in a browser

Configuring VS Code

Settings

Above, we added the packages `flake8` and `black` to our environment. The first one is a linter and the second a formatter, and they're here to help us follow python's general conventions.

Now we need to tell VS Code that we want to use them. We also need to configure VS Code to run our environment's python, not the global python installed in your machine.

Open any `*.py` file in your project, then press `CTRL+SHIFT+P` and search for `Python: select linter`. Choose `flake8` from the options.

Notice a new `.vscode` folder in your project with a `settings.json` inside. Open it and add the following:

```
"python.linting.flake8Args": [
        "--max-line-length=200" // customize this length to your taste afterwards
    ],
"python.formatting.provider": "black",
"[python]": {
    "editor.formatOnSave": true, //if this annoys you later, change it to false
},
```

Only one setting missing. You need to find the virtual environment's path and add it to the settings. Run this on the terminal:

```
poetry env info
```

Copy the path info and add it to the `settings.json` :

```
"python.pythonPath": "THE-PATH-YOU-COPIED/bin/python"
```

Debugger

Press `Run and Debug` icon on the left sidebar and click on `create a launch.json file` to open the debug configuration dialog, and select `Django` . It will then ask for the path to the `manage.py` file. This will depend on which folder you opened VS Code, in my case it's `${workspaceFolder}/ecommerce-app/ecommerce/manage.py`

A new file `launch.json` has been added to the `.vscode` folder with your debug configurations.

Debug test

1. Open `settings.py` and place a breakpoint on line 23
2. Press `F5` to start the debugger
3. The debugger should stop at line 23
4. Hover the `BASE_DIR` variable on line 16. If a bunch of info has appeared it means it's working
5. Press `SHIFT+F5` to stop the debugger and remove the breakpoint

Templates

Creating the template files

Inside the folder we called `store` we'll create two new folders called `templates/store` and inside them we'll create four html files:

- main.html - Template which all will inherit from
- store.html - Home page/store front with all products
- cart.html - Users shopping cart
- checkout.html - Checkout page

Leave the `main.html` file empty for now, and in the other three files add the following html:

```
store.html:
<h3>Store</h3>

checkout.html:
<h3>Checkout</h3>

cart.html:
<h3>Cart</h3>
```

Creating the views

We'll edit the file `store/views.py` to create our views. Add the following code to the file:

```
def store(request):
    context = {}
    return render(request, "store/store.html", context)


def cart(request):
    context = {}
    return render(request, "store/cart.html", context)


def checkout(request):
    context = {}
    return render(request, "store/checkout.html", context)
```

Creating the urls

Now we need to create some url paths to call these views.

Create a file called `urls.py` inside the `store` folder.

Add the following code to the file:

```
from django.urls import path

from . import views

urlpatterns = [
    # Leave as empty string for base url
    path("", views.store, name="store"),
    path("cart/", views.cart, name="cart"),
    path("checkout/", views.checkout, name="checkout"),
]
```

Base URLs Configuration

To connect our new urls we need to open up the `urls.py` file in the child `ecommerce` directory and "include" them:

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

urlpatterns = [
    path("admin/", admin.site.urls),
    path("", include("store.urls")),
]
```

Test it

Test if everything is good. Run `python manage.py runserver` on the terminal, open http://127.0.0.1:8000 in a browser and then check if the following are working:

- http://127.0.0.1:8000/ - should see the contents of store.html
- http://127.0.0.1:8000/cart - should see the contents of cart.html
- http://127.0.0.1:8000/checkout - should see the contents of checkout.html

Static files

Before we start working on our templates, let's configure the static files for images, css and js files.

In the root `ecommerce` directory create a folder called `static` .

Inside the new `static` folder, let's create a folder called `css` and another called `images` .

CSS file

Create a `main.css` file and add the following:

```
body {
    background-color: #999;
}
```

Now in order to see this in effect, we need to open up the `settings.py` file in the child `ecommerce` directory and add the following after `STATIC_URL = "static/"`

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

Add the following to the top of the `store.html` file:

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

Images

For now, just copy the `cart.png` image to the `static/images` folder.

And add the following to the end of the `store.html` file:

```
<img src="{% static 'images/cart.png' %}">
```

Test it

Test if everything is good. Run `python manage.py runserver` on the terminal, open http://127.0.0.1:8000 in a browser and then check if the following are working:

- http://127.0.0.1:8000/ - should see the grey background and the cart image

Main template

Before we get into designing our template, let's first remove that grey background and image we have.

Your css file should be blank now and your `store.html` file should only consist of:

```
{% load static %}

<h3>Store</h3>
```

Open `main.html` and create a standard html layout with the title “Ecom":

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ecomm</title>
</head>
<body>

</body>
</html>
```

Now let's add the static elements, as well as the Bootstrap framework to help us with the styling:

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ecomm</title>
    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
        rel="stylesheet"
        integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
        crossorigin="anonymous">
    <!-- Our CSS -->
    <link rel="stylesheet" type="text/css" href="{% static 'css/main.css' %}">
</head>
<body>
    <!-- Bootstrap JS -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"
        integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p"
        crossorigin="anonymous"></script>
</body>
</html>
```

We'll be creating the navigation shortly, but before let's define this file as the template from which all other pages will inherit from. Add the following just below the `body` tag:

```
<h3>Navbar Placeholder</h3>
<hr>
<div class="container">
    <br>
    {% block content %}

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

Now we want to have all the other files (`store.html` , `cart.html` , `checkout.html`) to inherit from `main.html` . Let's add the following code to each file**but ensure the text inside the h3 blocks is still unique**

```
{% extends 'store/main.html' %}
{% load static %}
{% block content %}
     <h3>Store</h3>
{% endblock content %}
```

Test it

Test if everything is good. Run `python manage.py runserver` on the terminal, open http://127.0.0.1:8000 in a browser and then check if the following are working:

- http://127.0.0.1:8000/ - should see the Navbar placeholder
- http://127.0.0.1:8000/cart - should see the Navbar placeholder
- http://127.0.0.1:8000/checkout - should see the Navbar placeholder

Navbar

Place the following code in the `main.html` instead of the `h3` and `hr` tags:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="{% url 'store' %}">Ecom</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse"
            data-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" href="{% url 'store' %}">Store</a>
                </li>
            </ul>
            <div class="mt-2">
                <a href="#" class="btn btn-warning">Login</a>
                <a href="{% url 'cart' %}">
                    <img id="cart-icon" src="{% static 'images/cart.png' %}">
                </a>
                <p id="cart-total">0</p>
            </div>
        </div>
    </div>
</nav>
```

Custom CSS

Add the following to the `main.css` file:

```
body {
    background-color: hsl(0, 0%, 98%);
}
h1,
h2,
h3,
h4,
h5,
h6 {
    color: hsl(0, 0%, 30%);
}
.box-element {
    box-shadow: hsl(0, 0%, 80%) 0 0 16px;
    background-color: #fff;
    border-radius: 4px;
    padding: 10px;
}
.thumbnail {
    width: 100%;
    height: 200px;
    -webkit-box-shadow: -1px -3px 5px -2px rgba(214, 214, 214, 1);
    -moz-box-shadow: -1px -3px 5px -2px rgba(214, 214, 214, 1);
    box-shadow: -1px -3px 5px -2px rgba(214, 214, 214, 1);
}
.product {
    border-radius: 0 0 4px 4px;
}
.bg-dark {
    background-color: #4f868c!important;
}
#cart-icon {
    width: 25px;
    display: inline-block;
    margin-left: 15px;
}
#cart-total {
    display: inline-block;
    text-align: center;
    color: #fff;
    background-color: #fb6464;
    width: 25px;
    height: 25px;
    border-radius: 50%;
    font-size: 14px;
}
.col-lg-4,
.col-lg-6,
.col-lg-8,
.col-lg-12 {
    margin-top: 10px;
}
.btn {
    border-radius: 0;
}
.row-image {
    width: 100px;
}
.form-field {
    width: 250px;
    display: inline-block;
```

```
    padding: 5px;
}
.cart-row {
    display: flex;
    align-items: flex-stretch;
    padding-bottom: 10px;
    margin-bottom: 10px;
    border-bottom: 1px solid #ececec;
}
.quantity {
    display: inline-block;
    font-weight: 700;
    padding-right: 10px;
}
.chg-quantity {
    width: 12px;
    cursor: pointer;
    display: block;
    margin-top: 5px;
    transition: .1s;
}
.chg-quantity:hover {
    opacity: .6;
}
.hidden {
    display: none!important;
}
```

Test it

Test if everything is good. Run `python manage.py runserver` on the terminal, open http://127.0.0.1:8000 in a browser and then check if the following are working:

- http://127.0.0.1:8000/ - should see the navbar and the new styles
- http://127.0.0.1:8000/cart - should see the navbar and the new styles
- http://127.0.0.1:8000/checkout - should see the navbar and the new styles

Pages

store.html

The entire page will consist of multiple rows with 3 columns per row that will hold product information. For now we will create just one row and add some data. Here's the steps needed to accomplish that.

1. Remove the `h3` tag
2. Copy the `placeholder.png` image to the `static/images` folder
3. Add this html to the page, between the `{% block content %}` tags:

```
<div class="row">
    <div class="col-lg-4">
        <img class="thumbnail" src="{% static 'images/placeholder.png' %}">
        <div class="box-element product">
            <h6><strong>Product 1</strong></h6>
            <hr>
            <button  class="btn btn-outline-secondary add-btn">Add to Cart</button>
            <a class="btn btn-outline-success" href="#">View</a>
            <h4 style="display: inline-block; float: right"><strong>$10</strong></h4>
        </div>
    </div>
    <div class="col-lg-4">
        <img class="thumbnail" src="{% static 'images/placeholder.png' %}">
        <div class="box-element product">
            <h6><strong>Product 2</strong></h6>
            <hr>
            <button  class="btn btn-outline-secondary add-btn">Add to Cart</button>
            <a class="btn btn-outline-success" href="#">View</a>
            <h4 style="display: inline-block; float: right"><strong>$15</strong></h4>
        </div>
    </div>
    <div class="col-lg-4">
        <img class="thumbnail" src="{% static 'images/placeholder.png' %}">
        <div class="box-element product">
            <h6><strong>Product 3</strong></h6>
            <hr>
            <button  class="btn btn-outline-secondary add-btn">Add to Cart</button>
            <a class="btn btn-outline-success" href="#">View</a>
            <h4 style="display: inline-block; float: right"><strong>$20</strong></h4>
        </div>
    </div>
</div>
```

cart.html

Here's the steps needed:

1. Remove the `h3` tag
2. Copy the `arrow-down.png` and `arrow-up.png` image to the `static/images` folder
3. Add this html to the page, between the `{% block content %}` tags:

```
<div class="row">
    <div class="col-lg-12">
        <div class="box-element">
            <a  class="btn btn-outline-dark" href="{% url 'store' %}">&#x2190; Continue Shopping</a>
            <br>
            <br>
            <table class="table">
                <tr>
                    <th><h5>Items: <strong>2</strong></h5></th>
                    <th><h5>Total:<strong> $40</strong></h5></th>
                    <th>
                        <a  style="float:right; margin:5px;" class="btn btn-success"
                            href="{% url 'checkout' %}">Checkout</a>
                    </th>
                </tr>
            </table>
        </div>
        <br>
        <div class="box-element">
            <div class="cart-row">
            <div style="flex:2"></div>
            <div style="flex:2"><strong>Item</strong></div>
            <div style="flex:1"><strong>Price</strong></div>
            <div style="flex:1"><strong>Quantity</strong></div>
            <div style="flex:1"><strong>Total</strong></div>
        </div>
        <div class="cart-row">
            <div style="flex:2">
                <img class="row-image" src="{% static 'images/placeholder.png' %}">
            </div>
            <div style="flex:2"><p>Product 3</p></div>
            <div style="flex:1"><p>$20</p></div>
            <div style="flex:1">
                <p class="quantity">2</p>
                <div class="quantity">
                    <img class="chg-quantity" src="{% static  'images/arrow-up.png' %}">

                    <img class="chg-quantity" src="{% static  'images/arrow-down.png' %}">
                </div>
            </div>
            <div style="flex:1"><p>$40</p></div>
        </div>
    </div>
</div>
```

checkout.html

Here's the steps needed:

1. Remove the `h3` tag
2. Add this html to the page, between the `{% block content %}` tags:

```
<div class="row">
    <div class="col-lg-6">
        <div class="box-element" id="form-wrapper">
            <form id="form">
                <div id="user-info">
                    <div class="form-field">
                        <input required class="form-control" type="text" name="name"
                            placeholder="Name..">
                    </div>
```

```
                    <div class="form-field">
                        <input required class="form-control" type="email" name="email"
                            placeholder="Email..">
                    </div>
                </div>
                <div id="shipping-info">
                    <hr>
                    <p>Shipping Information:</p>
                    <hr>
                    <div class="form-field">
                        <input class="form-control" type="text" name="address"
                            placeholder="Address..">
                    </div>
                    <div class="form-field">
                        <input class="form-control" type="text" name="city"
                            placeholder="City..">
                    </div>
                    <div class="form-field">
                        <input class="form-control" type="text" name="state"
                            placeholder="State..">
                    </div>
                    <div class="form-field">
                        <input class="form-control" type="text" name="zipcode"
                            placeholder="Zip code..">
                    </div>
                </div>
                <hr>
                <input id="form-button" class="btn btn-success btn-block" type="submit"
                    value="Continue">
            </form>
        </div>
        <br>
        <div class="box-element hidden" id="payment-info">
            <small>Paypal Options</small>
            <button id="make-payment" class="btn btn-info btn-block">Submit Order</button>
        </div>
    </div>
    <div class="col-lg-6">
        <div class="box-element">
            <a  class="btn btn-outline-dark" href="{% url 'cart' %}">&#x2190; Back to Cart</a>
            <hr>
            <h3>Order Summary</h3>
            <hr>
            <div class="cart-row">
                <div style="flex:2">
                    <img class="row-image" src="{% static 'images/placeholder.png' %}">
                </div>
                <div style="flex:2"><p>Product 3</p></div>
                <div style="flex:1"><p>$20.00</p></div>
                <div style="flex:1"><p>x2</p></div>
            </div>
            <h5>Items:   2</h5>
            <h5>Total:   $40</h5>
        </div>
    </div>
</div>
```

Data structure

Models

We'll start by setting up the models (database tables) needed for our application.

We'll do that in the `store/models.py` file.

Here's the code for our models:

```
...
from django.contrib.auth.models import User  # the django default user model

# Create your models here.


class Customer(models.Model):
    user = models.OneToOneField(User, null=True, blank=True, on_delete=models.CASCADE)
    name = models.CharField(max_length=200, null=True)
    email = models.CharField(max_length=200)

    def __str__(self):
        return self.name


class Product(models.Model):
    name = models.CharField(max_length=200)
    price = models.FloatField()
    digital = models.BooleanField(default=False, null=True, blank=True)
    # Todo: add image field

    def __str__(self):
        return self.name


class Order(models.Model):
    customer = models.ForeignKey(
        Customer, on_delete=models.SET_NULL, null=True, blank=True
    )
    date_ordered = models.DateTimeField(auto_now_add=True)
    complete = models.BooleanField(default=False)
    transaction_id = models.CharField(max_length=200, null=True)

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


class OrderItem(models.Model):
    product = models.ForeignKey(
        Product, on_delete=models.SET_NULL, null=True, blank=True
    )
    order = models.ForeignKey(Order, on_delete=models.SET_NULL, null=True, blank=True)
    quantity = models.IntegerField(default=0, null=True, blank=True)
    date_added = models.DateTimeField(auto_now_add=True)

    # no need for the method here, but we could do something like:
    # def __str__(self):
    #     return self.product.name


class ShippingAddress(models.Model):
    customer = models.ForeignKey(
        Customer, on_delete=models.SET_NULL, null=True, blank=True
    )
    order = models.ForeignKey(Order, on_delete=models.SET_NULL, null=True, blank=True)
    address = models.CharField(max_length=200, null=False)
    city = models.CharField(max_length=200, null=False)
    state = models.CharField(max_length=200, null=False)
    zip_code = models.CharField(max_length=200, null=False)
    date_added = models.DateTimeField(auto_now_add=True)

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

Migrations

Now we need to persist these models in the database. Run the following in the terminal:

```
python manage.py makemigrations
```

This will create the `store/migrations/0001_initial.py` file. Run this file with the following command:

```
python manage.py migrate
```

As you might have noticed, a lot more migrations have been ran (ours is just the last one):

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

That's because django comes builtin with an admin panel with authentication, as we'll see shortly.

Registering models

Finally, we need to let django know about our models because we want them available in the admin panel, so we'll add them to the `store/admin.py` file:

```
...
from . import models

# Register your models here.

admin.site.register(models.Customer)
admin.site.register(models.Product)
admin.site.register(models.Order)
admin.site.register(models.OrderItem)
admin.site.register(models.ShippingAddress)
```

Test

Check the following:

1. if the database has the tables

2. notice how all tables have a primary key field `id` that was added automatically for us
3. notice that all the tables have been prefixed with our app name: `store`

Adding user data

Let's start by creating a django `super user`, an admin that has access to everything. Run the following in a terminal:

```
python manage.py createsuperuser
```

Here's what you can type:

```
Username (leave blank to use 'rferreira'): superuser
Email address: superuser@mail.com
Password: #imtheBOSS!
Password (again): #imtheBOSS!
Superuser created successfully.
```

By the way, here's what happens if you try to insert a weak password, for ex: `123456`:

```
Email address: superuser@mail.com
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]:
```

Time to run our application again: `python manage.py runserver`

Adding product data

Let's use django's admin panel to add product data. Browse tohttp://127.0.0.1:8000/admin to meet this panel.

Login with the credentials of the super user we created:

```
username: superuser
password: #imtheBOSS!
```

So all our models are present, and we have an interface to perform CRUD operations on them, all without a single line of code.

Use the interface to add the following six products:

- name: Headphones
- price: 179.99
- digital: No

- name: Mount of Olives book
- price: 14.99
- digital: No

- name: Project source code

- price: 19.99
- digital: Yes

- name: Watch
- price: 259
- digital: No

- name: Shoes
- price: 89.99
- digital: No

- name: T-shirt
- price: 25.99
- digital: No

Rendering product data

Now we'll see how we can render this product data in our store. Let's start with the `store/views.py` file, and edit it like this:

```
...
from .models import * # another way to import the models (compare with admin.py)


def store(request):
    products = Product.objects.all()
    context = {"products": products}
    return render(request, "store/store.html", context)
...
```

Now let's edit our template, `store/templates/store/store.html`, and replace the `{% block content %}` with:

```
<div class="row">
    {% for product in products %}
    <div class="col-lg-4">
        <img class="thumbnail" src="{% static 'images/placeholder.png' %}">
        <div class="box-element product">
            <h6><strong>{{ product.name }}</strong></h6>
            <hr>
            <button  class="btn btn-outline-secondary add-btn">Add to Cart</button>
            <a class="btn btn-outline-success" href="#">View</a>
            <h4 style="float: right"><strong>${{ product.price|floatformat:2 }}</strong></h4>
        </div>
    </div>
    {% endfor %}
</div>
```

Test

Reload the store page to see if the changes have been applied.

File Handling

Everything is in place, except the product images. For that we'll have to config our application to handle file

uploads, storage and file url's. Let's solve this.

Add the image field in the model

Go to `store/models.py` and in the `Product` model replace the todo line with:

```
image = models.ImageField(null=True, blank=True)
```

As soon as you save this file, you'll see an error in your terminal:

```
ERRORS:
store.Product.image: (fields.E210) Cannot use ImageField because Pillow is not installed.
HINT: Get Pillow at https://pypi.org/project/Pillow/ or
run command "python -m pip install Pillow".
```

Pillow is a python package for everything related with image files, and we'll add it to our environment. Close the application server and type the following:

```
poetry add pillow
```

Start your server again (`python manage.py runserver`), and the error is gone.

Run migrations

Since we've added the image field to the model, we have to run the migration again to update de database:

```
python manage.py makemigrations
```

Notice the name of the migration file, it states what we've changed: `0002_product_image.py`

```
python manage.py migrate
```

Specify the storage folder for the uploads

Edit the `ecommerce\settings.py` file, and after the `STATICFILES_DIRS` line add:

```
MEDIA_ROOT = BASE_DIR / "static/images/"
```

Test

Go back to the admin panel, edit the products and add their images.

1. Check if the image files are in the `static/images` folder
2. Check if the image filenames have been added to the database

Specify the url to render the images on the page

Edit the `ecommerce\settings.py` file, and after the `MEDIA_ROOT` line add:

```
MEDIA_URL = "/images/"
```

Now we have to add this to the url patterns in `ecommerce/urls.py` (the comments are there just to help you):

```
from django.contrib import admin
from django.urls import path, include
# add the two imports below
from django.conf.urls.static import static
from django.conf import settings

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

# add the lines below
media_pattern = static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
urlpatterns.extend(media_pattern)
```

So we're specifying that all files in the `static/images` folder will be accessible like this: http://127.0.0.1:8000/images/filename.ext

Render images on templates

Let's render the product images in the `store/templates/store/store.html` . Replace `{% static 'images/placeholder.png' %}` with `{{ product.image.url }}` and you're good to go.

Test

Can you see the store page with all the images?

What if there's a product without an image?

Try to remove an image and refresh the page to see what happens. Yep, the page doesn't load at all.

We could set an if in the template to handle that, but let's solve it another way.

Open the `store/models.py` and add the following to the end of the `Product` class:

```
@property
def image_url(self):
    try:
        url = self.image.url
    except:
        self.image = "placeholder.png"
        url = self.image.url
    return url
```

So we're creating a new property that will try to grab the image url. If it fails, we'll reset the `image` property to the placeholder image and then grab its url. So if a product doesn't have an image it will appear with the placeholder.

Replace the template with this new property: `{{ product.image_url }}` instead of `{{ product.image.url }}`

Adding customer, order and order items data

We're going to add some data manually just to work on the cart template rendering. We'll work on adding new data through the store in the next module.

Let's go to the admin panel and add a customer:

```
User: superuser
Name: John Doe
Email: superuser@mail.com
```

Now add an order:

```
Customer: select Jonh Doe
Date: select today
Time: select now
Complete: leave unchecked
Transaction id: some random number like 3095850
```

And finally the order items:

```
Product: Mount of Olives book
Order: 1
Quantity: 2
```

```
Product: T-shirt
Order: 1
Quantity: 1
```

Rendering cart data

We'll be working on the `store/views.py` to prepare the data and on `store/templates/store/cart.html` to render it.

On `store/views.py`, add the following to the `cart()` function:

```
def cart(request):
    # the store will allow purchases whether the visitor is authenticated or not,
    # but we have to handle them differently.
    if request.user.is_authenticated:
        # get the customer
        customer = request.user.customer
        # get the incomplete order, or create a new one
        order, created = Order.objects.get_or_create(customer=customer, complete=False)
        # get the order items
        # we can query child objects by setting the parent value (order)
        # and the child in lowercase (orderitem)
        items = order.orderitem_set.all()
    else:
        items = []  # we'll handle this later
    context = {"items": items}
    return render(request, "store/cart.html", context)
```

On `store/templates/store/cart.html`, find the second `<div class="box-element">` and replace the second `<div class="cart-content">` and it's contents with:

```
{% for item in items %}
    <div class="cart-row">
        <div style="flex:2"><img class="row-image" src="{{ item.product.image_url }}"></div>
        <div style="flex:2"><p>{{ item.product.name }}</p></div>
        <div style="flex:1"><p>${{ item.product.price|floatformat:2 }}</p></div>
        <div style="flex:1">
            <p class="quantity">x{{ item.quantity }}</p>
            <div class="quantity">
                <img class="chg-quantity" src="{% static  'images/arrow-up.png' %}">

                <img class="chg-quantity" src="{% static  'images/arrow-down.png' %}">
            </div>
        </div>
        <div style="flex:1"><p>$40</p></div>
    </div>
{% endfor %}
```

Now it's only missing updating the sum of each item and the order total value and number of items.

As we did with the image url, we'll create new properties in the models classes. Add the following to the `OrderItem` class in `store/models.py` :

```
@property
def get_total(self):
    total = self.product.price * self.quantity
    return total
```

Back on `store/templates/store/cart.html` , update the `div` before `{% endfor %}` to:

```
<div style="flex:1"><p>${{ item.get_total|floatformat:2 }}</p></div>
```

And the following to the `Order` class in `store/models.py` :

```
@property
def get_cart_total(self):
    orderitems = self.orderitem_set.all()
    total = sum([item.get_total for item in orderitems])
    return total

@property
def get_cart_items(self):
    orderitems = self.orderitem_set.all()
    total = sum([item.quantity for item in orderitems])
    return total
```

We need to update the cart template, but we don't have the order object available, only the items, so update the `cart()` function in `store/views.py` :

```
# before
else:
    items = []  # we'll handle this later
context = {"items": items}

# after
else:
    items = []  # we'll handle this later
    order = {  # we need to set this to prevent errors when user is not logged in
        "get_cart_total": 0,
        "get_cart_items": 0,
    }
context = {"items": items, "order": order}
```

Now we're able to update the cart template. Update the `table` in `store/templates/store/cart.html`:

```
<th><h5>Items: <strong>{{ order.get_cart_items }}</strong></h5></th>
<th><h5>Total:<strong> ${{ order.get_cart_total|floatformat:2 }}</strong></h5></th>
```

Rendering checkout data

We'll update the `checkout()` function in `store/views.py` with the same code we used in the `cart()` function:

```
def checkout(request):
    # the store will allow purchases whether the visitor is authenticated or not,
    # but we have to handle them differently.
    if request.user.is_authenticated:
        # get the customer
        customer = request.user.customer
        # get the incomplete order, or create a new one
        order, created = Order.objects.get_or_create(customer=customer, complete=False)
        # get the order items
        # we can query child objects by setting the parent value (order)
        # and the child in lowercase (orderitem)
        items = order.orderitem_set.all()
    else:
        items = []  # we'll handle this later
        order = {  # we need to set this to prevent errors when user is not logged in
            "get_cart_total": 0,
            "get_cart_items": 0,
        }
    context = {"items": items, "order": order}
    return render(request, "store/checkout.html", context)
```

And update the template `store/templates/store/checkout.html`. Replace the `<div class="cart-row">` block and the two `<h5>` tags with:

```
{% for item in items %}
    <div class="cart-row">
        <div style="flex:2"><img class="row-image" src="{{ item.product.image_url }}"></div>
        <div style="flex:2"><p>{{ item.product.name }}</p></div>
        <div style="flex:1"><p>${{ item.product.price|floatformat:2 }}</p></div>
        <div style="flex:1"><p>x{{ item.quantity }}</p></div>
    </div>
{% endfor %}
<h5>Items:   {{ order.get_cart_items }}</h5>
<h5>Total:   ${{ order.get_cart_total|floatformat:2 }}</h5>
```

Cart functionality

Add to cart

For this part of the process we'll have to bring in javascript, which will be responsible for the transitions and events as the user interacts with the store.

Create a new `js` folder inside the `/static` folder, and inside the new folder create a new file `cart.js` .

Add this js file to the main template. Open `/store/templates/store/main.html` and, at the bottom, place the following before the closing `</body>` tag:

```
<script type="text/javascript" src="{% static 'js/cart.js' %}"></script>
```

We will also use jQuery, so paste the following after the `<!-- Jquery -->` comment:

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"
integrity="sha256-/xUj+3OJU5yExlq6GSYGSHk7tPXikynS7ogEvDej/m4="
crossorigin="anonymous"></script>
```

We'll work on what happens when visitors click the `Add to cart` button. One of the ways of doing this is by registering and `on click event` on a certain CSS class. So let's add the class `update-cart` to our `Add to cart` button, like so:

```
/store/templates/store/store.html
---
<button class="btn btn-outline-secondary add-btn cart-ops">Add to Cart</button>
```

So this class will be used as the trigger for the event. But if all products will have the same trigger, how do we know which product is the visitor clicking on? We have to add some html `data` attributes with information to help us out. Let's add them to the button, like so:

```
/store/templates/store/store.html
---
<button data-product="{{ product.id }}" data-action="add"
    class="btn btn-outline-secondary add-btn cart-ops">Add to Cart</button>
```

Let's type a test in our js file to check if everything is ok so far:

```
/* /static/js/cart.js */
$(function() {
    $(document).on('click', '.cart-ops', function() {
        var productId = $(this).data('product')
        var action = $(this).data('action')
        console.log('productId:', productId, 'action:', action)
    })
});
```

Test

1. Open the `Developer tools` of your browser (usually just right click the page and select `Inspect element` or `Developer tools`)
2. Select the `Console` tab
3. Click on the `Add to cart` button in the page
4. Returning to the `Developer tools` , you should see the product information printed on the console.

Before moving on, we must know if the user is logged in or not, because the cart is handled differently depending on this. Add the following to the top of the main template file, right before the closing `</head>` tag:

```
/store/templates/store/main.html
---
<script type="text/javascript">
    var user = '{{ request.user }}'
</script>
```

We're placing it there to make it accessible in any page of the application.

Let's deal with the logged in user first. We need to send the product data to a view to process the visitor's intent (add a product to the cart, remove a product from the cart, change quantities, etc).

Create a new function in `/store/views.py` to handle this:

```
from django.http import JsonResponse # add this to the top of the file
...
def updateItem(request):
    return JsonResponse('Item was added', safe=False)
```

Now in `/store/urls.py` let's add an url for our new view:

```
urlpatterns = [
    ...
    path("update_item", views.updateItem, name="update_item")
]
```

Test

1. Go to http://127.0.0.1:8000/update_item and check if you see the message

---

All good, so now we have to build the connection between javascript (frontend) and django (backend). Go back to the js file and below the `console.log` line add:

```
/* /static/js/cart.js */
if(user === 'AnonymousUser') {
    console.log('User is not authenticated.')
} else {
    updateUserOrder(productId, action)
}
```

And then define this function in a new line below the whole block:

```
/* /static/js/cart.js */
function updateUserOrder(productId, action) {
    console.log('User is authenticated.')

    $.ajax({
        context: this,
        type: "post",
        url: '/update_item',
        dataType: "json",
        data: JSON.stringify({
            'productId': productId,
            'action': action
        }),
        success: function(data) {
            console.log('Data:', data)
        },
        error: function(data) {
            console.log('Data:', data)
        }
    });
}
```

If you try to test this now, you'll get an error in the console: `POST http://localhost:8000/update_item 403 (Forbidden)`

That's because django has a security layer called CSRF (`Cross Site Request Forgery`) token in place, and we're not passing any token. These tokens are usually part of a form, but since we don't have a form let's apply the workaround from the django docs here.

First, add the code from the docs to the main template, inside the script tag we created for the user:

```
/store/templates/store/main.html
---
<script type="text/javascript">
    var user = '{{ request.user }}'

    function getToken(name) {
        var cookieValue = null;
        if (document.cookie && document.cookie !== '') {
            var cookies = document.cookie.split(';');
            for (var i = 0; i < cookies.length; i++) {
                var cookie = cookies[i].trim();
                if (cookie.substring(0, name.length + 1) === (name + '=')) {
                    cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                    break;
                }
            }
        }
        return cookieValue;
    }
    var csrftoken = getToken('csrftoken');
</script>
```

Second, add the token to the `updateUserOrder` function in the js file:

```
/* /static/js/cart.js */
function updateUserOrder(productId, action) {
    console.log('User is authenticated.')

    $.ajax({
        ...
        type: "post",
        headers: {"X-CSRFToken": csrftoken}, // add this line
        url: '/update_item',
        ...
    });
}
```

Test

1. Reload the site and click the `Add to product` button.
2. In the console you should now see `Data: Item was added`, which means that the ajax request reached the django view, which responded with that message.

Speaking of the view, that's what we need to change next to actually create the cart and add items to it. Change the function into this:

```
# /store/views.py
import json # add this to the top
...
def updateItem(request):
    data = json.loads(request.body)
    product_id = data["productId"]
    action = data["action"]

    customer = request.user.customer
    product = Product.objects.get(id=product_id)
    order, created = Order.objects.get_or_create(customer=customer, complete=False)
    orderItem, created = OrderItem.objects.get_or_create(order=order, product=product)

    if action == "add":
        orderItem.quantity += 1
    if action == "remove":
        orderItem.quantity -= 1

    orderItem.save()

    if orderItem.quantity <= 0:
        orderItem.delete()

    return JsonResponse("Item was added", safe=False)
```

Test

1. Click some `Add to cart` buttons and then go to the cart. You should have those products in your cart now.

Finally, we need to update the red notification in the upper right corner with the total number of items in our cart. For that we'll need to change both the view and js code.

Let's start with the view, where we have to rearrange the code like this (the commented lines are already there from the previous step):

```
# /store/views.py
...
def updateItem(request):
    response = {}
    try:
        # data = json.loads(request.body)
        # product_id = data["productId"]
        # action = data["action"]

        #customer = request.user.customer
        #product = Product.objects.get(id=product_id)
        #order, created = Order.objects.get_or_create(customer=customer, complete=False)
        #orderItem, created = OrderItem.objects.get_or_create(order=order, product=product)

        #if action == "add":
        #    orderItem.quantity += 1
        #if action == "remove":
        #    orderItem.quantity -= 1

        #orderItem.save()

        #if orderItem.quantity <= 0:
        #    orderItem.delete()

        response["status"] = True
        response["msg"] = "Item was added"
        response["cart_items"] = order.get_cart_items
    except Exception as e:
        response["status"] = False
        response["msg"] = e
        response["cart_items"] = 0

    return JsonResponse(response, safe=False)
```

And now the js file (the commented lines are already there from the previous step):

```
/* /static/js/cart.js */
function updateUserOrder(productId, action) {
    // console.log('User is authenticated.')

    // $.ajax({
    //     context: this,
    //     type: "post",
    //     headers: {"X-CSRFToken": csrftoken},
    //     url: '/update_item',
    //     data: JSON.stringify({
    //         'productId': productId,
    //         'action': action
    //     }),
    //     success: function(data) {
            if(data.status) {
                console.log('Data:', data.msg)
                $('#cart-total').text(data.cart_items)
            } else {
                console.log('Data:', data)
            }
        },
    //     error: function(data) {
    //         console.log('Data:', data)
    //     }
    // });
}
```

Test

1. Click some `Add to cart` buttons and check if the red notification increments.
2. Now navigate to another page. The notification is 0 again!

Well, looks like we're not done yet. We need to have the total number of items in every view of our store, so we'll have to pass that number in the context of very view. To prevent a lot of copy pasting, let's create a new function in our views file:

```
# /store/views.py
def get_cart_items(request):
    items = 0
    if request.user.is_authenticated:
        customer = request.user.customer
        order, created = Order.objects.get_or_create(customer=customer, complete=False)
        items = order.get_cart_items

    return items
```

Now find **all** `context` variables and add `"cartItems": get_cart_items(request)` to their dictionaries. Here's and example:

```
# /store/views.py

# before
context = {"products": products}
# after
context = {"products": products, "cartItems": get_cart_items(request)}
```

All that's missing is updating the template file:

```
/store/templates/store/main.html
---
<p id="cart-total">{{ cartItems }}</p>
```

Update cart

For this section we'll use a lot of what we made in the previous section. Let's take care of the quantity buttons in the cart.

First let's update the template to add the event handler and data attributes to the buttons:

```
/store/templates/store/cart.html
---
<img class="chg-quantity cart-ops" data-product="{{ item.product.id }}"
    data-action="add" src="{% static  'images/arrow-up.png' %}">
<img class="chg-quantity cart-ops" data-product="{{ item.product.id }}"
    data-action="remove" src="{% static  'images/arrow-down.png' %}">
```

Test

1. Try to increment/decrement the quantity of a product. It works (the red notification changes), but nothing else gets updated: the quantity, the total items nor the prices.

We need the page to reload, because we already have all this logic implemented. Let's edit the js file(the commented lines are already there from previous steps):

```
/* /static/js/cart.js */
function updateUserOrder(productId, action) {
    // console.log('User is authenticated.')

    // $.ajax({
    //     context: this,
    //     type: "post",
    //     headers: {"X-CSRFToken": csrftoken},
    //     url: '/update_item',
    //     data: JSON.stringify({
    //         'productId': productId,
    //         'action': action
    //     }),
    //     success: function(data) {
    //        if(data.status) {
    //            console.log('Data:', data.msg)
    //            $('#cart-total').text(data.cart_items) // this line doesn't matter anymore
                   location.reload()
    //        } else {
    //            console.log('Data:', data)
    //        }
    //    },
    //     error: function(data) {
    //         console.log('Data:', data)
    //     }
    // });
}
```

> This happens a lot during development: something we thought was the way to go turns out it's not. But be aware that the solution above might be inefficient if the page has more data and a reload is a costly operation.

Test

1. Increment/Decrement quantities and check if all the values in the page get updated accordingly.

Shipping address

We'll start off by adding some logic that will change the presentation of the page, because we only need a shipping address if the product is not digital.

Let's add a new property to our `Order` class in `store/models.py` :

```
@property
def shipping(self):
    shipping = False
    orderitems = self.orderitem_set.all()
    for item in orderitems:
        if item.product.digital == False:
            shipping = True
    return shipping
```

Now we have to add this property to the views, when the user is not authenticated, to prevent errors:

```
# /store/views.py

# change in cart and checkout functions
order = {  # we need to set this to prevent errors when user is not logged in
            "get_cart_total": 0,
            "get_cart_items": 0,
            "shipping": False,
        }
```

Finally, we need to hide the shipping address form in the template, right before the end of the file:

```
/store/templates/store/checkout.html
---
<script type="text/javascript">
    var shipping = '{{ order.shipping }}'
    if (shipping == 'False') {
        document.getElementById('shipping-info').innerHTML = ''
    }
</script>
{% endblock content %}
```

Test

1. Add/Remove products from the cart until you have only the digital product to check if the shipping address disappears from the checkout page.

Now we'll add a new feature to our checkout page: we'll show payment options to the user once he fills out the shipping details and clicks the `Continue` button.

First let's add the on submit event:

```
/store/templates/store/checkout.html
---
<script type="text/javascript">
    // var shipping = '{{ order.shipping }}'
    // if (shipping == 'False') {
    //     document.getElementById('shipping-info').innerHTML = ''
    // }

    var form = document.getElementById('form')
    form.addEventListener('submit', function(e) {
        e.preventDefault()
        console.log('Form Submitted..')
        document.getElementById('form-button').classList.add('hidden')
        document.getElementById('payment-info').classList.remove('hidden')
    })
</script>
```

Next up is the event handler for the `Make Payment` button which will actually process the order.

```
/store/templates/store/checkout.html
---
<script type="text/javascript">
    // var shipping = '{{ order.shipping }}'
    // if (shipping == 'False') {
    //     document.getElementById('shipping-info').innerHTML = ''
    // }

    // var form = document.getElementById('form')
    // form.addEventListener('submit', function(e) {
    //     e.preventDefault()
    //     console.log('Form Submitted..')
    //     document.getElementById('form-button').classList.add('hidden')
    //     document.getElementById('payment-info').classList.remove('hidden')
    // })

    document.getElementById('make-payment').addEventListener('click', function(e) {
        submitFormData()
    })
</script>
```

And here's the `submitFormData` function:

```
/store/templates/store/checkout.html
---
<script type="text/javascript">
    // var shipping = '{{ order.shipping }}'
    // if (shipping == 'False') {
    //     document.getElementById('shipping-info').innerHTML = ''
    // }

    // var form = document.getElementById('form')
    // form.addEventListener('submit', function(e) {
    //     e.preventDefault()
    //     console.log('Form Submitted..')
    //     document.getElementById('form-button').classList.add('hidden')
    //     document.getElementById('payment-info').classList.remove('hidden')
    // })

    // document.getElementById('make-payment').addEventListener('click', function(e) {
    //     submitFormData()
    // })

    function submitFormData() {
        console.log('Payment button clicked')
    }
</script>
```

Test

1. Fill out the form and click the `Continue` button. The form should disappear and show the payment button, and you should get a message in the console.
2. Click the `Make Payment` button. You should get a message in the console as well, coming from the `submitFormData` function.

Checkout form

There's still some more logic to apply to the form:

- if the user is logged in, we don't need the `Name` and `Email` fields

- if the user is logged in and we don't need shipping, hide form and show the payment info

Let's care of that, adding to the code we wrote in the previous step:

```
/store/templates/store/checkout.html
---
<script type="text/javascript">
    // var shipping = '{{ order.shipping }}'
    // if (shipping == 'False') {
    //     document.getElementById('shipping-info').innerHTML = ''
    // }

    if(user != 'AnonymousUser') {
        document.getElementById('user-info').innerHTML = ''
    }

    if (shipping == 'False' && user != 'AnonymousUser') {
        document.getElementById('form-wrapper').classList.add('hidden')
        document.getElementById('payment-info').classList.remove('hidden')
    }

    // var form = document.getElementById('form')
    // form.addEventListener('submit', function(e) {
    //     e.preventDefault()
    //     console.log('Form Submitted..')
    //     document.getElementById('form-button').classList.add('hidden')
    //     document.getElementById('payment-info').classList.remove('hidden')
    // })

    // document.getElementById('make-payment').addEventListener('click', function(e) {
    //     submitFormData()
    // })

    function submitFormData() {
        console.log('Payment button clicked')
    }
</script>
```

Test

1. Refresh the checkout page. The user/email fields should disappear since we're logged in.
2. Go back to the cart and remove all non-digital products. Returning to the checkout page all the fields should be gone now.

Now we're ready to really get the data from the forms and pass it to the backend to process the order. Let's keep on working in the js file:

```
/store/templates/store/checkout.html
---
<script type="text/javascript">
    // var shipping = '{{ order.shipping }}'
    var total = '{{ order.get_cart_total }}'
    // if (shipping == 'False') {
    ...

    function submitFormData() {
        console.log('Payment button clicked')

        var userFormData = {
            'name': null,
            'email': null,
            'total': total
        }
        if (user == 'AnonymousUser') {
            userFormData.name = form.name.value
            userFormData.email = form.email.value
        }

        var shippingFormData = {
            'address': null,
            'city': null,
            'state': null,
            'zipcode': null
        }
        if (shipping != 'False') {
            shippingFormData.address = form.address.value
            shippingFormData.city = form.city.value
            shippingFormData.state = form.state.value
            shippingFormData.zipcode = form.zipcode.value
        }

        console.log('Shipping:', shippingFormData)
        console.log('User:', userFormData)
    }
</script>
```

Process order

Let's work on the backend now, to handle the incoming data. As we did with the cart updates, we need to create a view and a url pattern.

Create a new function in `/store/views.py` :

```
def processOrder(request):
    return JsonResponse('Payment complete!', safe=False)
```

Now in `/store/urls.py` let's add an url for our new view:

```
urlpatterns = [
    ...
    path("process_order", views.processOrder, name="process_order")
]
```

Test

1. Go to http://127.0.0.1:8000/process_order and check if you see the message

All good, so now we have to build the connection between javascript (frontend) and django (backend). Go back to the js file and let's add the post request to the new view:

```
/store/templates/store/checkout.html
---
<script type="text/javascript">
    ...

    function submitFormData() {
        ...

        // console.log('User:', userFormData)

        var url = '/process_order'
        fetch(url, {
            method: 'POST',
            headers: {
                'Content-type':'application/json',
                'X-CSRFToken': csrftoken
            },
            body: JSON.stringify({'form': userFormData, 'shipping': shippingFormData}),
        })
        .then((response) => response.json())
        .then((data) => {
            console.log('Sucess:', data)
            alert('Transaction completed!')
            window.location.href = "{% url 'store' %}"
        })
    }
</script>
```

Test

1. Go to the checkout page and click all the way until the `Make Payment` button. You should get an alert saying `Transaction completed!` and upon closing the alert be redirected to the home page.

Let's continue on the backend:

```
import datetime # add this to the top of the file
...
def processOrder(request):
    transaction_id = datetime.datetime.now().timestamp()
    data = json.loads(request.body)

    if request.user.is_authenticated:
        customer = request.user.customer
        order, created = Order.objects.get_or_create(customer=customer, complete=False)
        total = float(data["form"]["total"])
        order.transaction_id = transaction_id

        if total == order.get_cart_total:
            order.complete = True
        order.save()

        if order.shipping == True:
            ShippingAddress.objects.create(
                customer=customer,
                order=order,
                address=data["shipping"]["address"],
                city=data["shipping"]["city"],
                state=data["shipping"]["state"],
                zip_code=data["shipping"]["zipcode"],
            )

    return JsonResponse("Payment complete!", safe=False)
```

Test

1. Go through the checkout process again, and everything should go out as the last test.
2. Check if the number in the red notification is now 0
3. Check the database to see if you have the shipping info in the table
4. Check the database to see if the order is marked as completed and a new order has automatically been created

Guest checkout

we've been using some javascript, but this section is the perfect example of why a fullstack developer needs to understand not only the backend language (python), but also frontend languages, like javascript, when we need the frontend to act as some sort of storage besides the usual web stuff.

What is guest checkout

In this part we'll implement a feature present in many ecommerce sites: the ability for a visitor to purchase products without having to register an account in the store. This is very important since a lot of potential customers will abandon the store if registration is mandatory, thus affecting conversion. The quickest and easiest the checkout process is, the highest the chance of purchase.

We'll do this using cookies, where we'll store the cart information for each visitor. Cookie information can only be stored as a string, so we'll build a JSON object containing the info and stringify it before saving the cookie; when we need that info, we'll reverse this process to access it.

Get ready

1. Make sure you're logged out (go to the admin panel to make sure)
2. Open the `Developer tools` of your browser (usually just right click the page and select `Inspect element`

or `Developer tools`)
3. Select the `Console` tab
4. Click on the `Add to cart` button in the page
5. Returning to the `Developer tools`, you should see the message `User is not authenticated`

Set cookies

We'll start off by creating a js function to get our cookie cart and some logic to create a new cart if it doesn't exist (find the commented line and paste after it):

```
/* /store/templates/store/main.html */
---
// var csrftoken = getToken('csrftoken');

function getCookie(name) {
    // split cookie string and get all name=value pairs in an array
    var cookieArr = document.cookie.split(";");

    // loop the array elements
    for(var i=0; i < cookieArr.length; i++) {
        var cookiePair = cookieArr[i].split("=");

        // remove whitespace from the beginning of the cookie name and compare it with given name
        if(name == cookiePair[0].trim()) {
            return decodeURIComponent(cookiePair[1]);
        }
    }

    // return null if not found
    return null;
}

var cart = JSON.parse(getCookie('cart'));

if (cart == undefined) {
    cart = {};
    document.cookie = 'cart=' + JSON.stringify(cart) + ";domain=;path=/";
    console.log('Cart created!');
}

console.log('Cart', cart);
```

Test

1. Open the `Developer tools` of your browser (usually just right click the page and select `Inspect element` or `Developer tools`)
2. Select the `Console` tab
3. Refresh the page
4. Returning to the `Developer tools`, you should see the message `Cart created!`

Now we'll implement adding/removing products to/from our cookie cart (the commented lines are already there from a previous step):

```
/* /static/js/cart.js */
---
// if(user === 'AnonymousUser') {
//            console.log('User is not authenticated.') // delete this line
               addCookieItem(productId, action)
//        } else {
//            updateUserOrder(productId, action)
//        }
//    })
// });

function addCookieItem(productId, action) {
    console.log('User is not authenticated.')

    if(action == 'add') {
        if(cart[productId] == undefined) {
            cart[productId] = {'quantity' : 1}
        } else {
            cart[productId]['quantity'] += 1
        }
    }

    if(action == 'remove') {
        cart[productId]['quantity'] -= 1

        if(cart[productId]['quantity'] <= 0) {
            delete cart[productId]
        }
    }

    document.cookie = 'cart=' + JSON.stringify(cart) + ";domain=;path=/"
    location.reload()
    console.log('Cart', cart)
}
```

Test

1. Open the `Developer tools` of your browser (usually just right click the page and select `Inspect element` or `Developer tools`)
2. Select the `Console` tab
3. Add products
4. Returning to the `Developer tools` , you should see those products and quantities added to the cart.
5. Also check the `cart` cookie in the `Application` tab, under `Cookies` .

Render cart total

So we can add or remove products from the cart, but the cart total notification is not being updated. Let's take care of that now. We'll just fix the cart page for now and eventually get to all the other pages.

Let's edit the cart view, where we have to rearrange the code like this (the commented lines are already there from previous steps):

```
# /store/views.py
...
def get_cart_items(request, cart={}):
   # items = 0
   # if request.user.is_authenticated:
   #     customer = request.user.customer
   #     order, created = Order.objects.get_or_create(customer=customer, complete=False)
   #     items = order.get_cart_items
    else:
        for i in cart:
            items += cart[i]["quantity"]

   # return items
...
def cart(request):
   # # the store will allow purchases whether the visitor is authenticated or not,
   # # but we have to handle them differently.
   # if request.user.is_authenticated:
   #     # get the customer
   #     customer = request.user.customer
   #     # get the incomplete order, or create a new one
   #     order, created = Order.objects.get_or_create(customer=customer, complete=False)
   #     # get the order items
   #     # we can query child objects by setting the parent value (order)
   #     # and the child in lowercase (orderitem)
   #     items = order.orderitem_set.all()
        context = {"items": items, "order": order, "cartItems": get_cart_items(request)}
   # else:
        try:
            cart = json.loads(request.COOKIES["cart"])
        except Exception:
            cart = {}
   #     items = []  # we'll handle this later
   #     order = {  # we need to set this to prevent errors when user is not logged in
   #         "get_cart_total": 0, "get_cart_items": 0, "shipping": False}
        context = {"items": items, "order": order, "cartItems": get_cart_items(request, cart)}

   # return render(request, "store/cart.html", context)
```

Test

1. Go to the cart page. You should see the red notification in the top right corner updated with the total products in the cart.
2. Add more products and check if it updates.

Build order

We'll continue from the previous step and now we'll actually build the order with products, prices, etc.

```
# /store/views.py
...
def cart(request):
    ...
   # else:
   #     try:
   #         cart = json.loads(request.COOKIES["cart"])
   #     except:
   #         cart = {}
   #     items = []  # we'll handle this later
   #     order = {  # we need to set this to prevent errors when user is not logged in
   #         "get_cart_total": 0, "get_cart_items": 0, "shipping": False}

        for i in cart:
            try:
                product = Product.objects.get(id=i)
                total = (product.price * cart[i]["quantity"])

                order["get_cart_total"] += total
                order["get_cart_items"] += cart[i]["quantity"]

                item = {
                    "product": {
                        "id": i,
                        "name": product.name,
                        "price": product.price,
                        "image_url": product.image_url,
                    },
                    "quantity": cart[i]["quantity"],
                    "get_total": total,
                }

                items.append(item)
            except Exception:
                pass

   #     context = {"items": items, "order": order, "cartItems": get_cart_items(request, cart)}
```

Test

1. Refresh the cart page. You should see all the products listed and the totals.
2. Add/remove products using the arrows and see if everything updates accordingly.

Ok, one final detail regarding the shipping information. We need to check if the products are digital or not to enable/disable showing the shipping information. Let's add it to the previous code.

```
# /store/views.py
...
def cart(request):
    ...
    # item = {
    #     "product": {
    #         "id": i,
    #         "name": product.name,
    #         "price": product.price,
    #         "image_url": product.image_url,
    #     },
    #     "quantity": cart[i]["quantity"],
    #     "get_total": total,
    # }

    # items.append(item)

      if product.digital is False:
            order["shipping"] = True
    # except Exception:
    #     pass
```

cookiecart() function

A lot of the functionalities we've built above have to be replicated in other views such as checkout, so we would have to duplicate the code in all those views. That's not very good and it doesn't follow the DRY principle, so we'll extract the code into a function and then apply it everywhere we need.

We'll create a `utils.py` file with that function and then update all the views.

```
# /store/utils.py
import json
from .models import *

def cookieCart(request):
    return {}
```

Now we'll import that new function into our `views.py` file:

```
# /store/views.py
---
# from .models import *  # another way to import the models (compare with admin.py)
from .utils import cookieCart

# def get_cart_items(request, cart={}):
```

Let's build the function by copying and adapting the code from the previous steps

```python
# /store/utils.py
---
def cookieCart(request):
    try:
        cart = json.loads(request.COOKIES["cart"])
    except Exception:
        cart = {}
    items = []
    order = {"get_cart_total": 0, "get_cart_items": 0, "shipping": False}
    cartItems = order["get_cart_items"]

    for i in cart:
        try:
            cartItems += cart[i]["quantity"]

            product = Product.objects.get(id=i)
            total = product.price * cart[i]["quantity"]

            order["get_cart_total"] += total
            order["get_cart_items"] += cart[i]["quantity"]

            item = {
                "product": {
                    "id": i,
                    "name": product.name,
                    "price": product.price,
                    "image_url": product.image_url,
                },
                "quantity": cart[i]["quantity"],
                "get_total": total,
            }

            items.append(item)

            if product.digital is False:
                order["shipping"] = True
        except Exception:
            pass

    return {"items": items, "order": order, "cartItems": cartItems}
```

Now we're ready to apply this function to our views. Since we'll be doing a lot of changes, to make it easier here's the three functions we'll rewrite:

```
# /store/views.py
---
# from .utils import cookieCart

def store(request):
    if request.user.is_authenticated:
        customer = request.user.customer
        order, created = Order.objects.get_or_create(customer=customer, complete=False)
        cartItems = order.get_cart_items
    else:
        cookieData = cookieCart(request)
        cartItems = cookieData["cartItems"]

    products = Product.objects.all()
    context = {"products": products, "cartItems": cartItems}
    return render(request, "store/store.html", context)


def cart(request):
    if request.user.is_authenticated:
        customer = request.user.customer
        order, created = Order.objects.get_or_create(customer=customer, complete=False)
        items = order.orderitem_set.all()
        cartItems = order.get_cart_items
    else:
        cookieData = cookieCart(request)
        cartItems = cookieData["cartItems"]
        order = cookieData["order"]
        items = cookieData["items"]

    context = {"items": items, "order": order, "cartItems": cartItems}
    return render(request, "store/cart.html", context)


def checkout(request):
    if request.user.is_authenticated:
        customer = request.user.customer
        order, created = Order.objects.get_or_create(customer=customer, complete=False)
        items = order.orderitem_set.all()
        cartItems = order.get_cart_items
    else:
        cookieData = cookieCart(request)
        cartItems = cookieData["cartItems"]
        order = cookieData["order"]
        items = cookieData["items"]

    context = {"items": items, "order": order, "cartItems": cartItems}
    return render(request, "store/checkout.html", context)

# def updateItem(request):
```

Test

1. Browse through the main page, cart and checkout. All the data should be correct by now: cart totals, cart details, prices, product images, etc.

cartData() function

The `views.py` should be as slim as possible, and all the logic should be moved to another file. We'll continue the process started in the previous function and remove even more logic from this file.

We'll create a new function `cartData()` to hold that logic.

```
# /store/utils.py
# import json
# from .models import *

  def cartData(request):
      return {}

# def cookieCart(request):
```

Now we'll import that new function into our `views.py` file:

```
# /store/views.py
---
# from .models import *  # another way to import the models (compare with admin.py)
from .utils import cookieCart, cartData (replaces the existing cookieCart)

# def get_cart_items(request, cart={}):
```

Let's build the function by copying and adapting the code from the previous steps

```
# /store/utils.py
---
def cartData(request):
    if request.user.is_authenticated:
        customer = request.user.customer
        order, created = Order.objects.get_or_create(customer=customer, complete=False)
        items = order.orderitem_set.all()
        cartItems = order.get_cart_items
    else:
        cookieData = cookieCart(request)
        cartItems = cookieData["cartItems"]
        order = cookieData["order"]
        items = cookieData["items"]
    return {"items": items, "order": order, "cartItems": cartItems}
```

Now we're ready to apply this function to our views. Since we'll be doing a lot of changes, to make it easier here's the three functions we'll rewrite:

```
# /store/views.py
---
# from .utils import cartData

def store(request):
    data = cartData(request)
    cartItems = data["cartItems"]

    products = Product.objects.all()

    context = {"products": products, "cartItems": cartItems}
    return render(request, "store/store.html", context)

def cart(request):
    data = cartData(request)
    cartItems = data["cartItems"]
    order = data["order"]
    items = data["items"]

    context = {"items": items, "order": order, "cartItems": cartItems}
    return render(request, "store/cart.html", context)

def checkout(request):
    data = cartData(request)
    cartItems = data["cartItems"]
    order = data["order"]
    items = data["items"]

    context = {"items": items, "order": order, "cartItems": cartItems}
    return render(request, "store/checkout.html", context)
```

Much cleaner, right?

Test

1. Browse through the main page, cart and checkout. All the data should be correct by now: cart totals, cart details, prices, product images, etc.

Checkout

In this final step we'll take care of the checkout process for guest users.

We'll start off by clearing the cart after a guest finishes the purchase. Add the following to the `checkout.html` file:

```
/* /store/templates/store/checkout.html */
---
// alert('Transaction completed!')

   cart = {}
   document.cookie = 'cart=' + JSON.stringify(cart) + ';domain=;path=/'

// window.location.href = "{% url 'store' %}"
// })
```

Now let's add the logic to actually submit the order. First we'll create a new function in `utils.py` :

```
# /store/utils.py
---
def guestOrder(request, data):
    name = data["form"]["name"]
    email = data["form"]["name"]

    cookieData = cookieCart(request)
    items = cookieData["items"]

    # using the get or create method is cool because this way we know all the orders
    # made by the same guest,
    # and if he actually creates an account we'll have all it's previous orders
    customer, created = Customer.objects.get_or_create(email=email)
    customer.name = name
    customer.save()

    order = Order.objects.create(customer=customer, complete=False)

    for item in items:
        product = Product.objects.get(id=item["product"]["id"])

        orderItem = OrderItem.objects.create(
            product=product, order=order, quantity=item["quantity"]
        )

    return customer, order
```

And now we'll use that function in our rewritten `processOrder` function:

```
# /store/views.py
---
# from .utils import cartData, guestOrder
---
def processOrder(request):
#    transaction_id = datetime.datetime.now().timestamp()
#    data = json.loads(request.body)
#
#    if request.user.is_authenticated:
#        customer = request.user.customer
#        order, created = Order.objects.get_or_create(customer=customer, complete=False)
    else:
        customer, order = guestOrder(request, data)

    total = float(data["form"]["total"])
    order.transaction_id = transaction_id
    if total == order.get_cart_total:
        order.complete = True
    order.save()

    if order.shipping == True:
        ShippingAddress.objects.create(
            customer=customer,
            order=order,
            address=data["shipping"]["address"],
            city=data["shipping"]["city"],
            state=data["shipping"]["state"],
            zip_code=data["shipping"]["zipcode"],
        )

#    return JsonResponse("Payment complete!", safe=False)
```

Test

1. Login to the admin panel in http://127.0.0.1:8000/admin using the credentials:

```
username: superuser
password: #imtheBOSS!
```

Go to Orders and Customers and take note of how many you have so far.

1. Log out and go through the checkout process as a guest user.
2. Return to the admin panel. You should have a new customer and order created.

Appendix

Environment files

It's not a good practice to leave all the app settings in the `settings.py` file. Things like database credentials or the app's secret key being some of them.

Other settings, although not posing any security threat, are also nice to have in a separate file.

Other frameworks use environment files (.env files), and you can also use that in Django. We'll install a package called `django-environ` that makes the whole process easy.

Add/Install the package

On a terminal with your virtual environment activated, type:

```
poetry add django-environ
```

Import and initialize environ in `settings.py`

```
import environ
# Initialise environment variables
env = environ.Env()
env.read_env()
```

Create your .env and env.example file

In the same directory as `settings.py`, create a file called `.env` and `env.example`.

The example file will be commited to git to serve as a reference.

Declare your environment variables in .env

```
SECRET_KEY=h^z13$qr_s_wd65@gnj7a=xs7t05$w7q8!x_8zsld#
DEBUG=True

#DATABASE_NAME=postgresdatabase
#DATABASE_USER=root
#DATABASE_PASS=supersecretpassword
```

Add your .env file to .gitignore

If you don’t have a .gitignore file already, create one at the project root.

Make sure your .env file is included. Here's an example:

```
# Poetry
poetry.lock

# VS Code
.vscode/

# DB
ecommerce/db.sqlite3

# Django Project
ecommerce/ecommerce/.env
ecommerce/ecommerce/__pycache__/

# Store App
ecommerce/store/__pycache__/
ecommerce/store/migrations/__pycache__/
```

Replace all references to your environment variables in `settings.py`

Example:

```
# before
SECRET_KEY = 'h^z13$qr_s_wd65@gnj7a=xs7t05$w7q8!x_8zsld#'

# after
SECRET_KEY = env(‘SECRET_KEY’)
```

Executing tests

Every app created in Django contains a file called `tests.py` by default. You can put your tests here or you can create a sub-directory called `tests` .

When your project grows and you want to better organise your tests, you can put them into a separate directory. If you go down this route, make sure you do these four things:

- Name the directory `tests`
- Add a file called `__init__.py` into the tests folder and all of its sub-folders. The file can be empty, it just needs to exist.
- Begin each file name with `test_` . This is so Django knows where to look for tests.
- Remove the original `tests.py` from the app’s directory.

> If using poetry, you can also delete the `tests` directory created by poetry.

Your first test

Testing that two plus two equals four might seem pointless but it’s a quick way to check that our test suite is structured correctly.

Do the steps above and create a new file called `test_one.py` inside the `tests` folder.

Go to `test_one.py` and type the following:

```
from django.test import TestCase


def add_two_numbers(a, b):
    return a + b


class TestExample(TestCase):

    def test_add_two_numbers(self):
        self.assertEqual(add_two_numbers(2, 2), 4)
```

To run your test, type in the terminal:

```
python manage.py test store/tests
```

If you want to run only the tests inside a specific file, run:

```
python manage.py test store.tests.test_one
```

Add tests to VS Code debugger

Just append the following to the `configurations` list in the `launch.json` file of the `.vscode` folder:

```
{
    "name": "Run Tests",
    "type": "python",
    "request": "launch",
    "program": "${workspaceFolder}/ecommerce/manage.py",
    "args": [
        "test"
    ],
    "django": true
}
```

Using class-based views instead of functions

If you prefer working with object oriented programming you can use classes instead of functions to specify your views.

How to implement a class-based view

Imagine that you have the following function-based view that will handle GET or POST requests to the same view:

```
def simple_function_based_view(request):
    if request.method == 'GET':
        ...  # code to process a GET request
    elif request.method == 'POST':
        ...  # code to process a POST request
```

Here's the implementation of that function in a class:

```
from django.views import View

class SimpleClassBasedView(View):
    def get(self, request):
        ...  # code to process a GET request

    def post(self, request):
        ...  # code to process a POST request
```

If you use classes, your `urls.py` file also needs to change:

```
urlpatterns = [
    # the function-based view is added as the second param
    url('function/', views.simple_function_based_view),
    # the as_view method is called on the class-based view and
    # the result is the value of the second param
    url('class/', views.SimpleClassBasedView.as_view()),
]
```

The class-based views (view classes that extend the `View` class) get built-in responses to unsupported methods (POST, PUT, PATCH, etc.) and get support for the OPTIONS HTTP method too.

All that is required to support other HTTP methods is to implement the same named method on the view class.

Creating a view that will render an html template

The view classes that extend the `View` class usually return an `HTTPResponse()`. Throughout the course we never used that, instead we rendered an html template with some context. To achieve this, we're better off creating our views by extending the `TemplateView` class instead.

Here's an example:

```
from datetime import datetime
from django.http import HttpResponse
from django.views.generic import TemplateView  # import the TemplateView class

class SimpleClassBasedView(TemplateView):  # extend from TemplateView
    template_name = 'store/test.html'  # add a template_name attribute

    # change the get method to get_context_data
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['now'] = datetime.now() # add more data to context if needed
        return context
```

The `urls.py` file would remain unchanged:

```
urlpatterns = [
    # the function-based view is added as the second param
    url('function/', views.simple_function_based_view),
    # the as_view method is called on the class-based view and
    # the result is the value of the second param
    url('class/', views.SimpleClassBasedView.as_view()),
]
```

Creating a view that will render all static pages

Your website might have pages that are purely static, such as the privacy policy or the about us page. If using classes, you can create a single class that will handle all these pages.

First you create the view class:

```
from django.http import HttpResponse
from django.views import View
from django.http import Http404 # import this to return a 404 not found error

class StaticView(View):
    static_html = None

    def get(self, request):
        if not self.static_html:
            raise Http404
        html = open(self.static_html).read()
        return HttpResponse(html)
```

And then in your urls file:

```
url('test/', views.StaticView.as_view(static_html='store/about_us.html'))
```

Integrating Django with a legacy database

While Django is best suited for developing new applications, it's quite possible to integrate it into legacy databases. Django includes a couple of utilities to automate as much of this process as possible.

Give Django your database parameters

Edit the `DATABASES` setting and assign values to the following keys:

- NAME
- ENGINE
- USER
- PASSWORD
- HOST
- PORT

Auto-generate the models

Django comes with a utility that can create models by introspecting an existing database. You can view the output by running this command:

```
python manage.py inspectdb
```

Save this as a file by using standard Unix output redirection:

```
python manage.py inspectdb > models.py
```

By default, `inspectdb` creates unmanaged models. That is, `managed = False` in the model's Meta class tells Django not to manage each table's creation, modification, and deletion:

```
class Person(models.Model):
    id = models.IntegerField(primary_key=True)
    first_name = models.CharField(max_length=70)
    class Meta:
       managed = False
       db_table = 'CENSUS_PERSONS'
```

If you do want to allow Django to manage the table's lifecycle, you'll need to change the managed option above to `True` (or remove it because `True` is its default value).

Finally, you'll need to put the `models.py` file inside an app and run the migrations:

```
python manage.py migrate
```

Outputting csv with Django

Just a simple snippet you can use in your views to output a csv file for download:

```
import csv
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(
        content_type='text/csv',
        headers={'Content-Disposition': 'attachment; filename="somefilename.csv"'},
    )

    writer = csv.writer(response)
    writer.writerow(['First row', 'Foo', 'Bar', 'Baz'])
    writer.writerow(['Second row', 'A', 'B', 'C'])

    return response
```

Other Django packages and resources

Below is a list of other packages and resources you might find interesting:

- django-compressor - combines and minifies CSS and JS files -https://github.com/django-compressor/django-compressor
- django-allauth - authentication, registration, account management as well as 3rd party (social) account authentication - https://github.com/pennersr/django-allauth
- django-debug-toolbar - the name says it all -https://github.com/jazzband/django-debug-toolbar
- django-seed - automatically populate your database with fake data -https://github.com/brobin/django-seed
- Cookiecutter Django - a framework for jumpstarting production-ready Django projects quickly - https://github.com/cookiecutter/cookiecutter-django
- Visit https://www.djangotemplatetagsandfilters.com for a quick and easy way to browse all the available tags and filters.
- Visit https://awesomedjango.org for a lot more.