DJANGO MADE EASY

SECOND EDITION

BUILD AND DEPLOY RELIABLE DJANGO APPLICATIONS

PETER VOUGHT

DJANGO MADE EASY

SECOND EDITION

BUILD AND DEPLOY RELIABLE DJANGO APPLICATIONS

PETER VOUGHT

Django made Easy

second edition

Build and deploy reliable Django applications

Peter Vought

Table of Contents

Introduction

“Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It’s free and open source.”

Those are some of the first words from Django documentation. It can also be described as a very capable Python web framework with a relatively low learning curve. It can be used for building applications ranging from small blogs to large scale applications, as it is a very robust and scalable framework. It also comes with “batteries included”, meaning some of the functionality is already provided by default without the need to reinvent the wheel while avoiding security pitfalls at the same time.

I intentionally wrote this book in an easy-to-follow way so that even a complete beginner can feel comfortable following along. While putting this project together, I occasionally had to compromise between being easy to follow even for complete beginners while going into technical details at the same time to provide value for more experienced users.

This book takes a project-oriented approach to teaching. Instead of exploring different aspects of the Django framework in isolation, we will be building and deploying real-world, fully functional e-commerce application to see how all Django pieces come together. The final product of our effort is available at https://finesauces.store/ .

Throughout this book, we will cover the Django framework's essential aspects: URLs, views, models, templates, sessions, authentication, and much more. To conclude our project, we will go through a step-by-step deployment process to provide you with full hands-on development experience. By the end of this book and project, you will be well suited to start creating your own projects and make your ideas become a reality!

I did my best to make the content as easy to grasp as possible. If you, however, get stuck or face any issues along the way, please don't hesitate to reach out to me on facebook at *@peter.vought.author* or on my email *peter@vought.tech* .

Happy coding!

1 Initial setup

The following section serves as a quick guide to setup and installing all required dependencies for local development. Django, being a Python web framework, needs an active Python installation to run. The latest available Django version, 3.1, supports by default Python versions 3.6 and newer. Throughout this book, we will be using Python version 3.8. If you already have Python3 installed on your system, feel free to move to the next section, where we set up our database, control version system, and virtual environment.

Let's confirm your active Python3 installation by opening the terminal and typing the *python3 (python* for Windows*)* command. Your version might be different, but if you receive a message similar to this:

```
terminal
~$ python3
Python 3.8.5 (v3.8.5:580fbb018f, Jul 20 2020, 12:11:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

it means Python3 is already set up on your system, and you are ready to get started. However, please visit the official Python site https://www.python.org/downloads/ and follow an installation guide specific to your system if you encounter an error message.

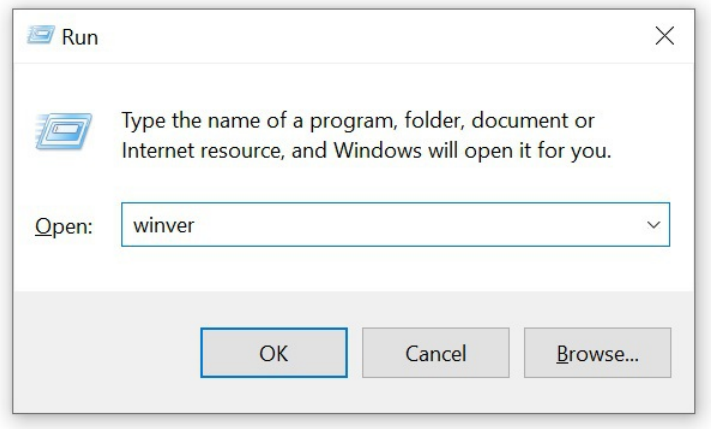
1.1 Windows Subsystem for Linux (WSL 2)

This is an optional chapter for Windows users. Since we plan on deploying this project on Ubuntu 20.04 LTS server and we want to mimic our future production environment on our local machine to the maximum extent, it might be a reasonable to set up Linux environment on the Windows machine. That is where Windows Subsystem for Linux comes in. As per Microsoft documentation [1] *"The Windows Subsystem for Linux lets developers run a GNU/Linux environment -- including most command-line tools, utilities, and applications -- directly on Windows, unmodified, without the overhead of a traditional virtual machine or dualboot setup."* , meaning we can take advantage of the Linux environment without having to reinstall our OS in order to switch to Linux distro or having to dualboot. The following guide is based on the official Microsoft documentation [2] .

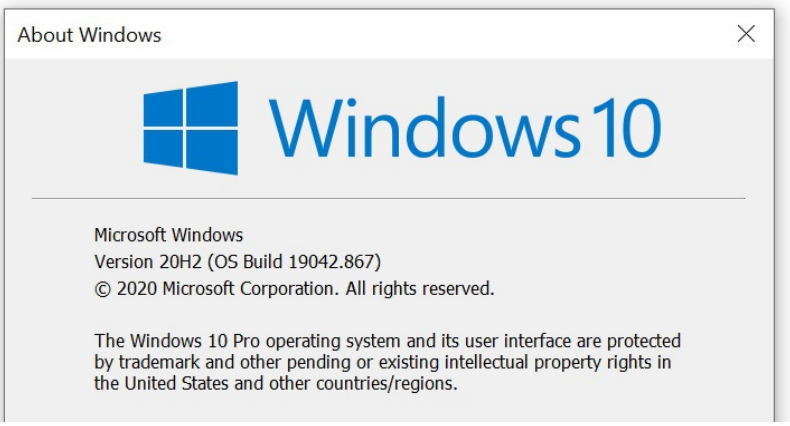
1.1.1 Checking requirements for running WSL 2

In order to run WSL2 on Windows 10, we need to make sure our OS meets the version criteria - **Version 1903** or higher, with **Build 18362** or higher [3] .

In order to verify version and build number, press **Windows logo key + R** and type *winver* to the window that appears:



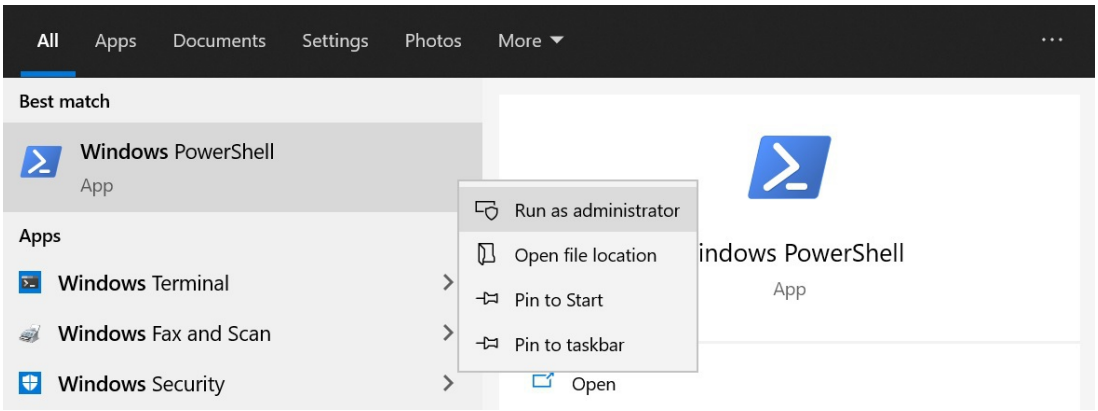
The following window will be invoked, revealing essential Windows 10 data:



If your *Version* or *Build* happens to be below required value, please make sure to update your OS.

1.1.2 Enabling the Windows Subsystem for Linux

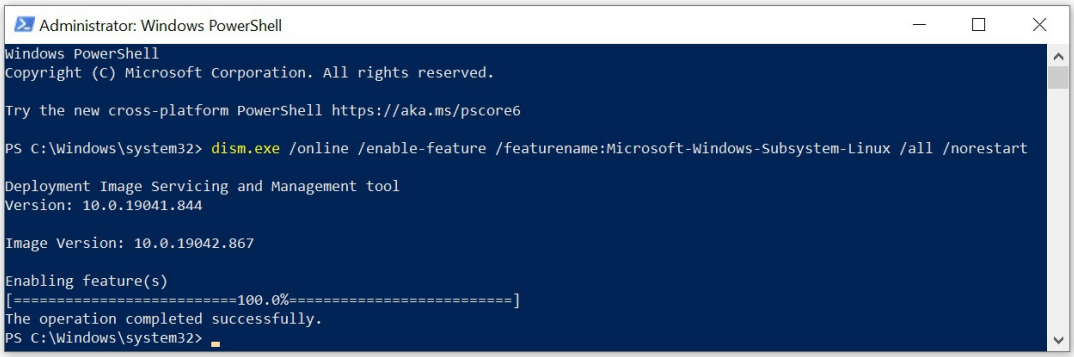
Before installing any Linux distributions on Windows, we must first enable the "*Windows Subsystem for Linux*" optional feature. To do so, Open PowerShell as Administrator:



and run the following command:

```PowerShell
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
```

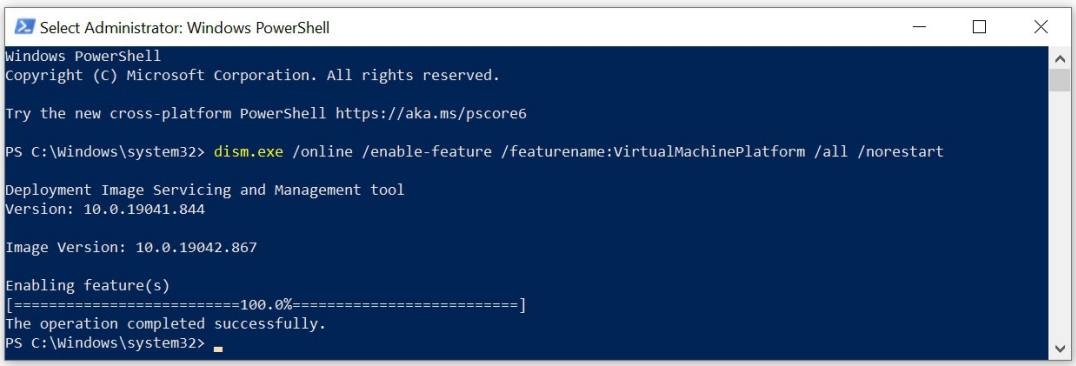
You should see the similar output:



1.1.3 Enabling Virtual Machine feature

Before installing WSL 2, we must enable the **Virtual Machine Platform** optional feature. Make sure you are running PowerShell as Administrator and use the command:

```PowerShell
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```



If you happen to face any issues with this step, please review the following documentation [4] .

Restart your machine before moving to the next section.

1.1.4 Downloading the Linux kernel update package

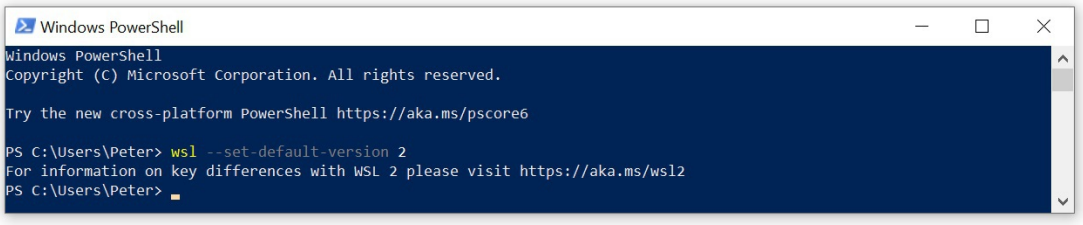
Before downloading and installing our WSL 2, we need to download the latest kernel update package.

Visit https://wslstorestorage.blob.core.windows.net/wslblob/wsl_update_x64.msi to download the file. Once download finishes, run the package by double-clicking on it. You might be prompted for elevated permissions, select *"yes"* to approve the installation.

1.1.5 Setting WSL 2 as our default version

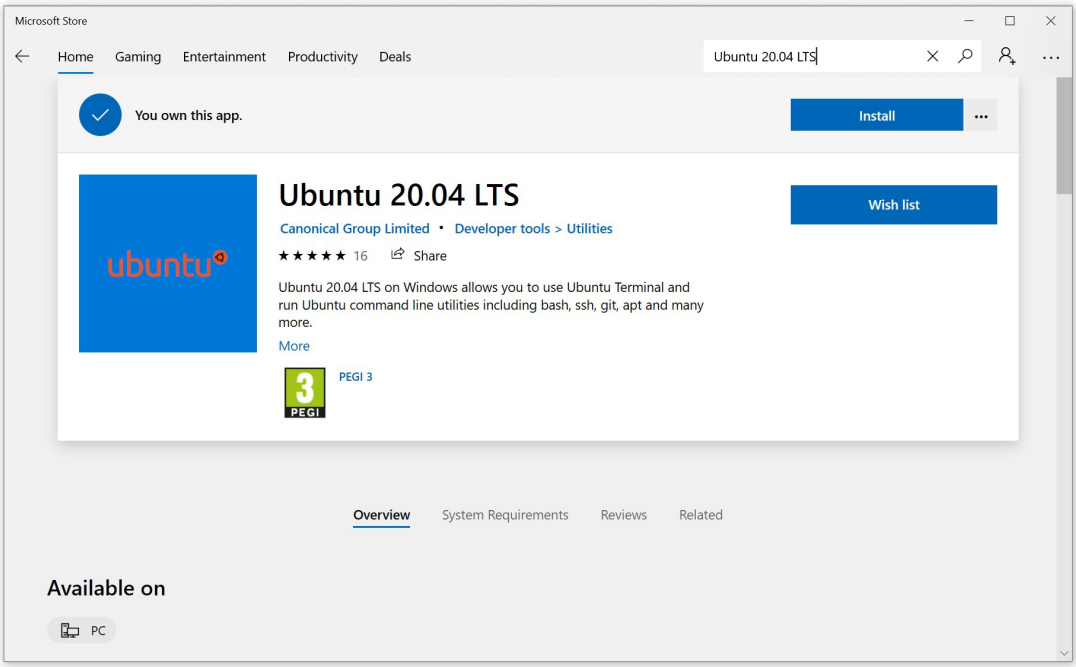
Open your *PowerShell* (don't need to run it as Administrator) and run the following command to set WSL 2 as the default version:

```PowerShell
wsl --set-default-version 2
```

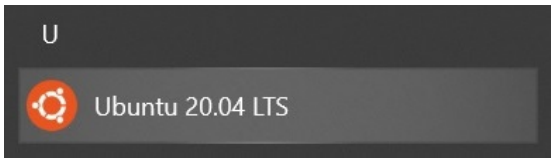


1.1.6 Installing Ubuntu 20.04 LTS

Open *Microsoft Store* and search for *Ubuntu 20.04 LTS* :



Click on *Install* . Once the download finishes, you can launch it either from the *Store* , or *Start* menu:



When you launch a newly installed Linux distribution, a console window will open and you'll be asked to wait for a minute or two for files to de-compress and be stored on your PC. All future launches should take less than a second.

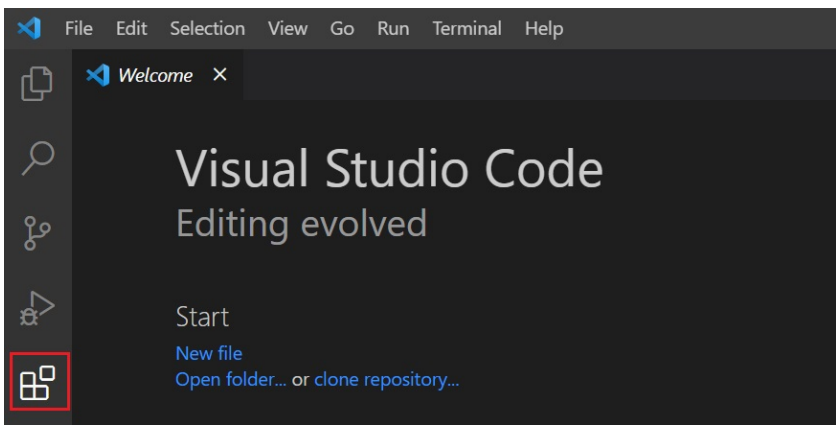
You will be asked to create a user account and provide password for your Ubuntu 20.04 LTS installation. Follow the prompts to finish the installation.

1.1.7 WSL 2 in VS Code

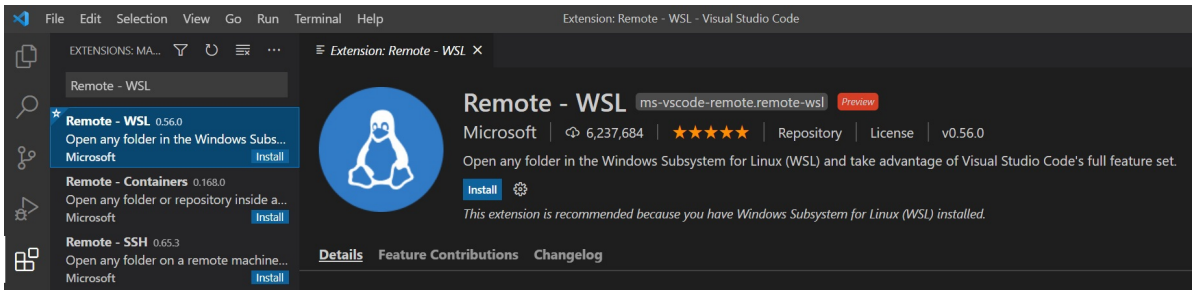
Visual Studio Code (VS Code) is a free coding editor that is my go-to option when it comes to Web, Python and Django development. It also provides seamless integration for the WSL 2, which we just installed. Visit the following site https://code.visualstudio.com/ to download and install the editor.

To access the WSL 2 from the VS Code, we will need to download the *Remote – WSL* extension from the Extension Marketplace.

Once inside the editor, click on the Marketplace icon:

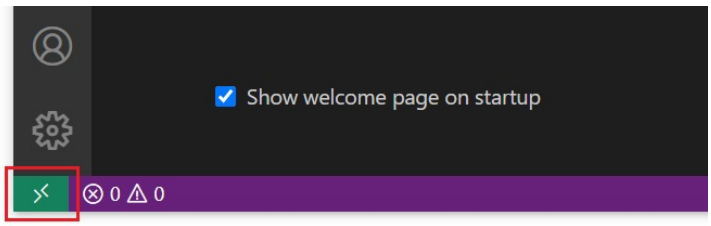


Type in the *Remote – WSL* name to the search field and select the appropriate option:

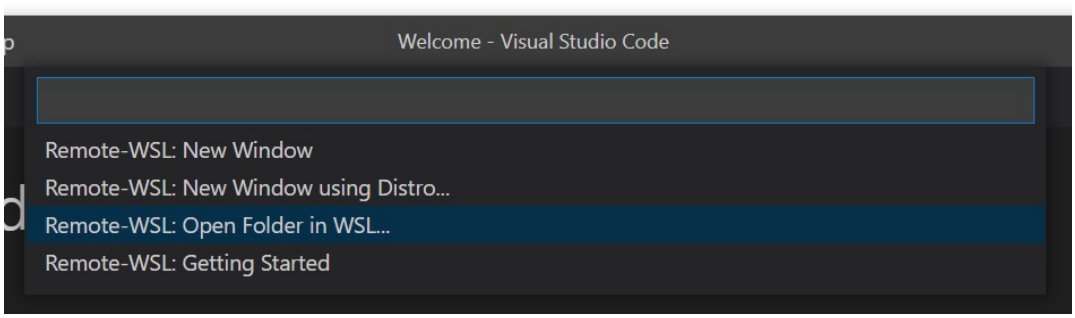


Click on *Install* . Once the installation finishes, close the extension window.

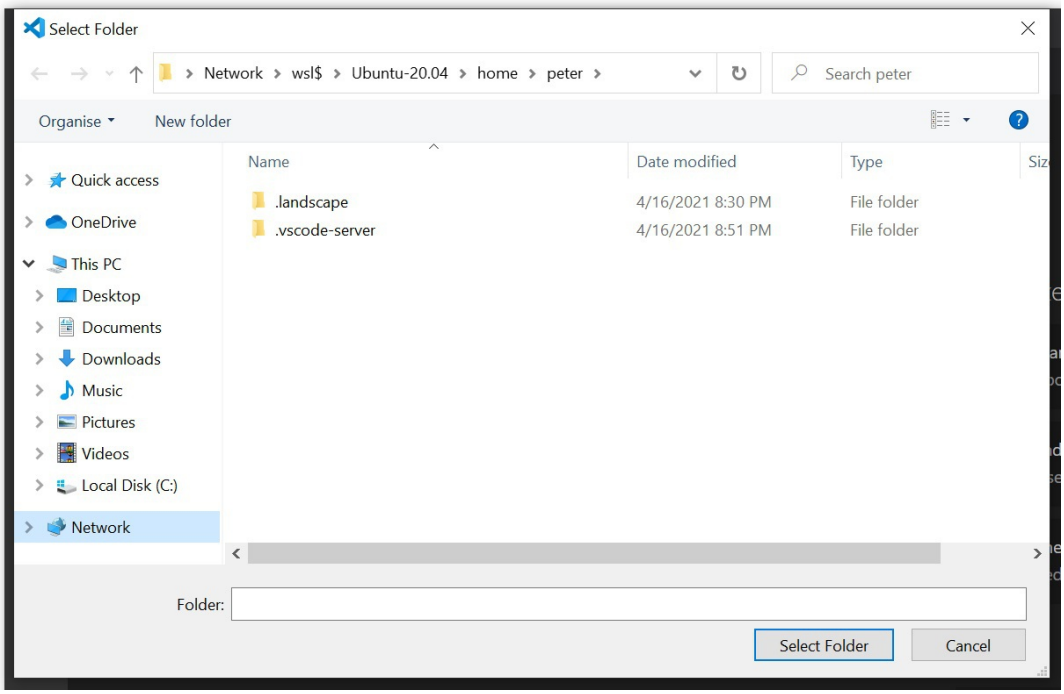
We can now go ahead and establish connection to our WSL 2. Navigate to the bottom left corner and look for the green icon:



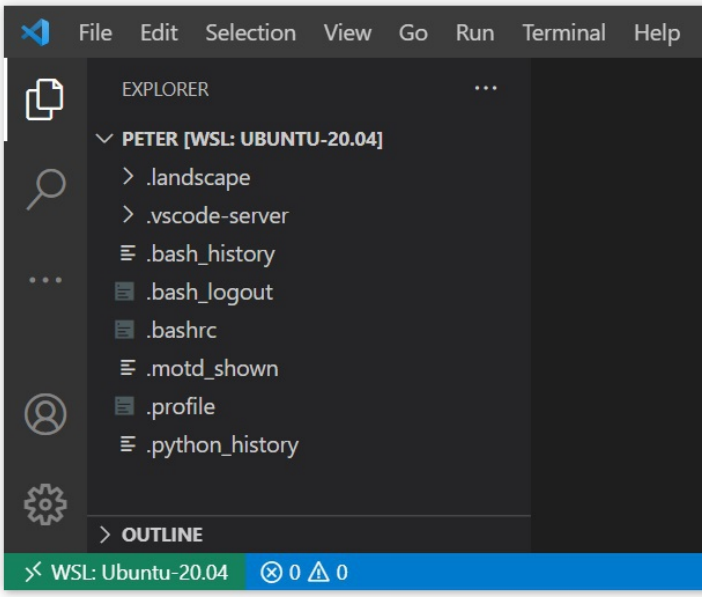
Once you click on the icon, dropdown menu will appear on the top of the editor. Select *Remote-WSL: Open Folder in WSL* option:



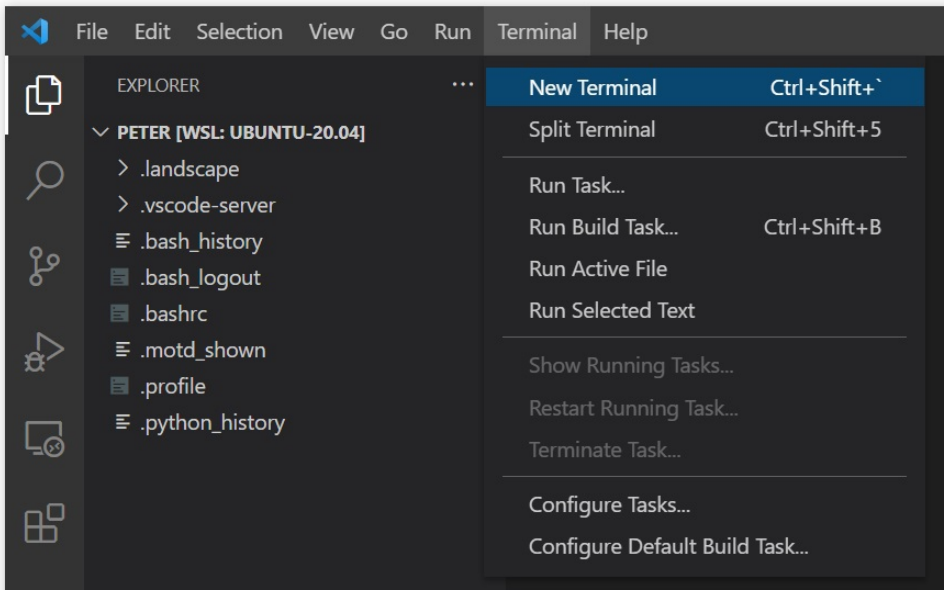
You will be then prompted to select the WSL 2 home folder:



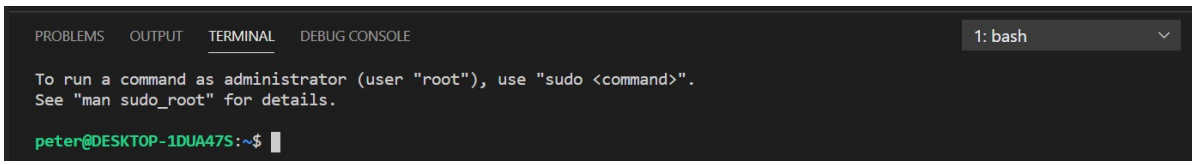
Click on *Select Folder* . After a brief moment, WSL 2 will be loaded. You should see very similar folder and file structure to this:



To make sure everything is up and running properly, let's invoke WSL 2 terminal. Click on *Terminal* in the top navbar and select *New Terminal* :



New section will appear at the bottom. Focus on the *Terminal* tab:



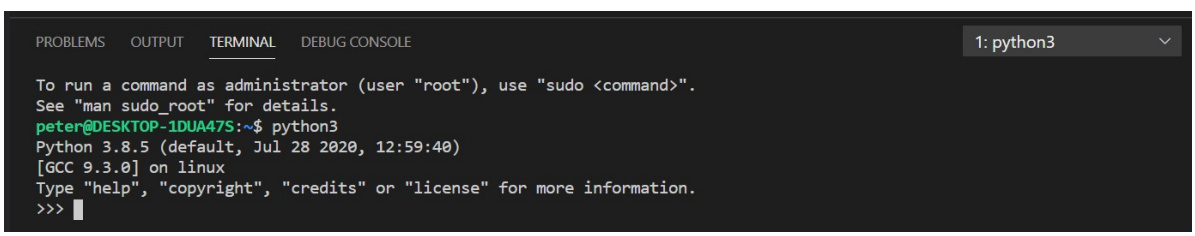
Let's get our WSL 2 up to date by downloading the latest updates available by running the following commands:

```
terminal
~$ sudo apt update
~$ sudo apt upgrade
```

You will be asked about installing new packages. Select Y to confirm:

```
terminal
After this operation, 175 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

A this stage, we can also check if any active Python 3 installation is present on the system. Type in the *python3* command. Python 3 prompt should be displayed:



Great job! We managed to set up a Linux distribution that is now completely integrated with the Windows OS. Developing web projects in WSL 2 is a great way to make sure no unexpected issues arise during the deployment process, since we are deploying to essentially identical environment.

1.2 PostgreSQL

When we create our initial Django project, it is by default configured to work with the SQLite database system. This database is sufficient for simple local development purposes as it is small, fast, and file-based. However, it is advised for the production environment to utilize a more robust and full-featured database, such as PostgreSQL, MariaDB, MySQL, or Oracle, which are officially supported by Django. Since we want to mimic our future production environment on our local machine as much as possible, we will utilize the PostgreSQL system, which is very popular amongst Django developers, from the beginning of our development process.

For installation instructions specific to your OS, please review the documentation available at https://www.postgresql.org/download/ .

If you are running Debian or Ubuntu based Linux distribution (including WSL 2), you can go ahead and use the following command to download and install PostgreSQL database:

terminal
```
~$ sudo apt install postgresql postgresql-contrib
```

Select Y to confirm installation:

terminal
```
After this operation, 121 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

After the installation finishes, we need to start the database server with command:

terminal
```
~$ sudo pg_ctlcluster 12 main start
```

To log in to the database session, we need to run:

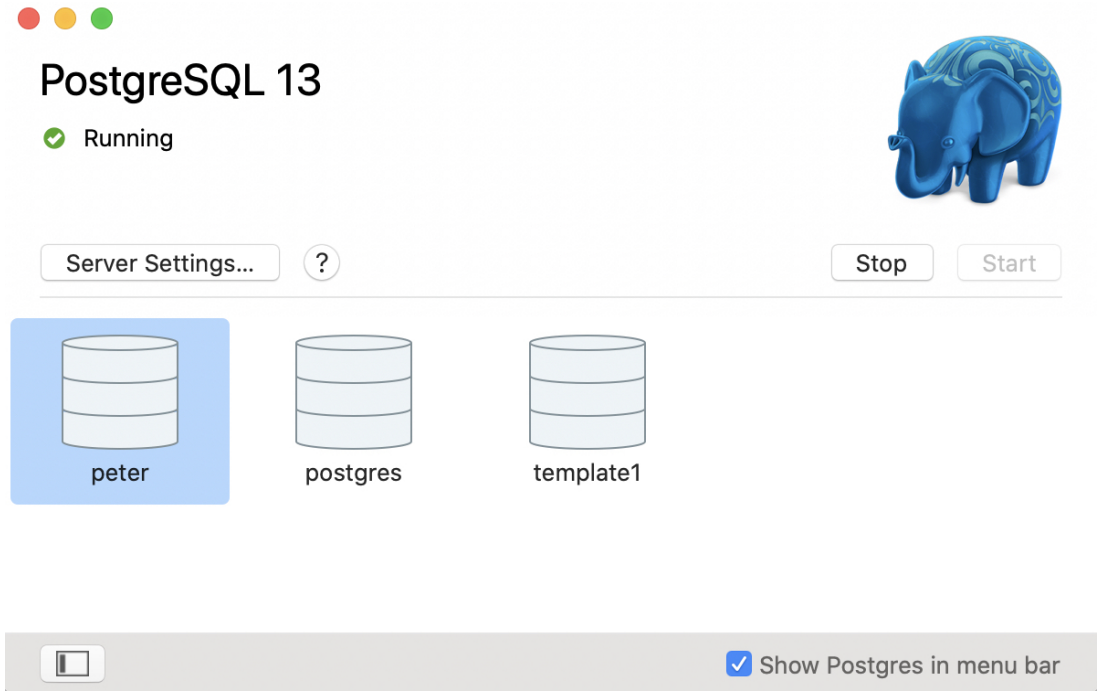
terminal
```
~$ sudo -u postgres psql
```

which states that we want to log in as *postgres* user into *psql* process, which represents a terminal-based front-end to PostgreSQL server. While logged in the Postgres session, a command prompt will change from *$* to *#* .

In case you are running macOS, you can go ahead and install it by downloading and setting up *Postgres.app* [5] . To download the app, visit https://postgresapp.com/downloads.html . After installation process finishes, run the app. You might have to click on "Initialize" to create a new server instance. By doing so, you will create PostgreSQL server with the following default settings:

Host	localhost
Port	5432
User	*your system user name*
Database	*same as user*
Password	*none*
Connection URL	postgresql://localhost

Now we need to set up a database for our project. While in Postgres app, double click on your default database to log into Postgres session:



Let's create a database called *finesauces* for our project by using the following command:

```
postgres terminal
postgres=# CREATE DATABASE finesauces;
```

Upon a successful database creation, you should see the *CREATE DATABASE* message. Let's create a dedicated user for our database (replace * with your password):

```
postgres terminal
postgres=# CREATE USER finesaucesadmin WITH PASSWORD '********';
```

To follow a Django recommendation regarding PostgreSQL configuration [6] , we will modify a couple of connection parameters for our *finesaucesadmin* user. This will slightly speed up the database operations so that the correct values do not have to be queried and set each time a connection is established. We will set encoding to UTF-8, which is expected by Django, transaction isolation to 'read committed', where the user will see only data committed before the query begins. Lastly, we set the timezone to 'UTC':

```
postgres terminal
postgres=# ALTER ROLE finesaucesadmin SET client_encoding TO 'utf8';
postgres=# ALTER ROLE finesaucesadmin SET default_transaction_isolation TO 'read committed';
postgres=# ALTER ROLE finesaucesadmin SET timezone TO 'UTC';
```

Now let's grant our user unrestricted access to administer the *finesauces* database:

```
postgres terminal
postgres=# GRANT ALL PRIVILEGES ON DATABASE finesauces TO finesaucesadmin;
```

We can quit the PostgreSQL session by typing:

```
postgres terminal
postgres=# \q
```

1.3 Git

Git [7] is an indispensable part of modern software development. It is a distributed version control system for tracking changes throughout the development process. It enables us to keep records of any changes made to our code with specific details like what and when was modified. It also allows us to revert back to previous working versions in case anything breaks or gets deleted!

The installation process is pretty straightforward. If you are running Debian or Ubuntu-based Linux distribution, you can use *sudo apt-get install git* command to install Git. On macOS, a recommended way is to use *brew install git* command.

Once the Git installation is done, we need to perform a one-time system setup to set a user name and email that will be associated with all our Git commits. Open the terminal and type in the following commands (replace the placeholders with your desired values):

```
terminal
~$ git config --global user.name "<your name>"
~$ git config --global user.email "<your email>"
```

Name and email settings can always be changed by re-running these commands.

1.4 Virtual environment

Virtual environments represent an essential element of Python programming. They are an isolated container which includes all the software dependencies for a given project. This is important because software like Python and Django is installed by default as system-wide. This might cause a severe problem when you want to work on multiple projects on the same computer. What if one project depends on Django 3.1, but a project from the previous year requires Django 2.2? Without virtual environments, this scenario becomes very difficult to manage.

Each virtual environment has its own Python binary and can have its own independent set of installed Python packages in its *site* directories. Using the Python *venv* module to create isolated Python environments allows us to use different package versions for different projects, which is far more practical than installing Python packages system-wide.

On macOS and Linux, you can install the *virtualenv* module by running:

```
terminal
~$ python3 -m pip install --user virtualenv
```

Before initializing our virtual environment, let's go ahead and create the main project folder called *finesauces* by using the *mkdir* command:

```
terminal
~$ mkdir finesauces
```

and move inside this directory:

```
terminal
~$ cd finesauces
```

Now that we are inside our project directory, we can create and initialize the virtual environment:

```
terminal
~/finesauces$ python3 -m venv env
```

This will create *env/* directory, including a Python environment. Any Python libraries we install while our virtual environment is active will go into the *env/lib/python3.8/site-packages* directory.

Activate your virtual environment by using the following command:

```
terminal
~/finesauces$ source env/bin/activate
```

If your environment got activated successfully, your terminal would also include parenthesis with the environment name within:

```
terminal
(env) ~/finesauces$
```

A virtual environment can be deactivated anytime by the *deactivate* command. For more information regarding *venv* functionality, please refer to the documentation available at https://docs.python.org/3/library/venv.html .

2 Starting our e-commerce project

In the previous chapter, we went through the necessary initial one-time setup. We installed the database system PostgreSQL, the versioning system Git and created our virtual environment, where our Django project will reside. Let's pick up where we left off and start by installing the Django framework, which is available as a Python package and therefore can be installed in any Python environment. Run the following command to install Django with *pip* :

```
terminal
(env) ~/finesauces$ pip install "Django==3.1.*"
```

Django will be installed in the Python *site-packages/* directory of our virtual environment.

2.1 Creating a Django project

Let's proceed to create our Django project called *finesauces project* with the following command, which creates our initial project structure:

```
terminal
(env) ~/finesauces$ django-admin startproject finesauces_project .
```

It's worth pausing here to explain why you might want to add the period . to the *startproject* command. If you were to run *django-admin startproject finesauces_project* without a period at the end, then by default Django would create the following directory structure:

```
finesauces
├── env
└── finesauces_project
    ├── finesauces_project
    │   ├── __init__.py
    │   ├── asgi.py
    │   ├── settings.py
    │   ├── urls.py
    │   └── wsgi.py
    └── manage.py
```

See how it creates a new directory *finesauces_project* and then within it *manage.py* and another *finesauces_project* directory? I prefer to avoid this extra layer in the folder structure, as it seems redundant to me. By running *django-admin startproject finesauces_project* . with a period at the end, we tell Django to initialize the project in the current directory and create the following folder structure:

```
finesauces
├── env
├── finesauces_project
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

This folder structure has no impact on the functionality or performance of our application whatsoever. It is only my preferred approach to structuring new projects. Let's briefly explore these files:

- ***manage.py*** - This is a command-line utility used to interact with our project. It is a thin wrapper around the *django-admin.py* tool.
- ***finesauces_project*** - This is our project directory, which consists of the following files:
    - *__init__.py* - Empty file that tells Python to treat the *finesauces_project* directory as a Python module.
    - *asgi.py* - This is the configuration to run our project as ASGI, the emerging Python standard for asynchronous web servers and applications.
    - *settings.py* - This file specifies settings and configurations for our project and contains initial default settings.
    - *urls.py* - Contains our URL patterns, which are mapped to specific views
    - *wsgi.py* - This is the configuration to run our project as a **Web Server Gateway Interface(WSGI)** application.

2.1.1 Running our local development server

To confirm everything was set up and installed correctly, let's try running a local Django development server by using the following command:

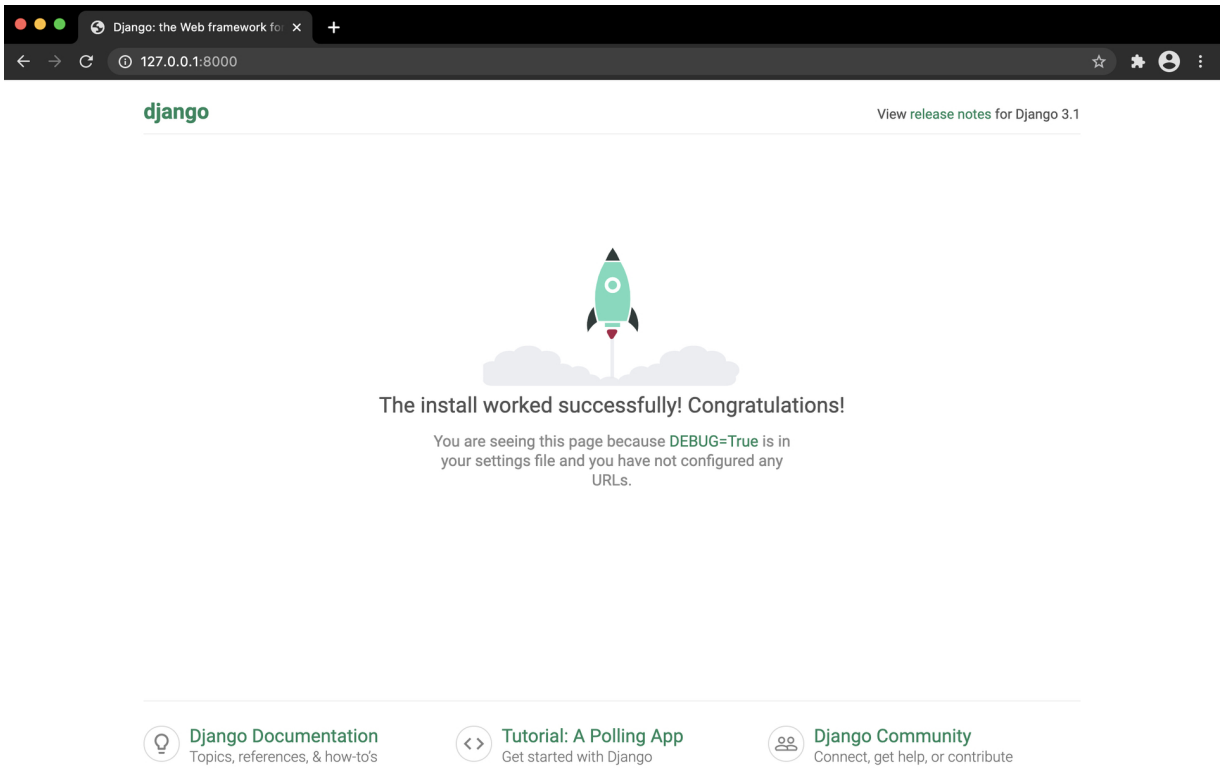
```
terminal
(env) ~/finesauces$ python manage.py runserver
```

If everything is set up properly, you should see a message similar to this in your terminal:

```
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.
January 02, 2021 - 20:50:54
Django version 3.1.4, using settings 'finesauces_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

You may notice a warning regarding unapplied migrations. We will take care of that shortly. Now open http://127.0.0.1:8000/ in your browser. You should see a welcome page stating that the project is successfully running, as shown in the following screenshot:



If you take a closer look at your terminal now, you will see the *GET* request performed by the browser:

terminal
```
[02/Jan/2021 20:51:09] "GET / HTTP/1.1" 200 16351
```

Each HTTP request is logged in the console by the development server. Any error that occurs while running the development server will also appear in the console.

We made sure our setup is correctly configured by running our local development server and visiting this welcome screen. Go ahead and stop the server by using *CONTROL* + *C* command.

2.2 Updating project settings

Now that our initial project structure is created, we need to let Django know that we will be using PostgreSQL database instead of default SQLite. We will achieve this by updating project settings located in the *finesauces_project* folder inside the *settings.py* file. Let's open this file and find the following code section:

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

Replace it with:

/finesauces/finesauces_project/settings.py
```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'finesauces',
        'USER': 'finesaucesadmin',
        'PASSWORD': '********',
        'HOST': 'localhost'
    }
}
```

We updated the database ENGINE to *postgresql* , database NAME to *finesauces* , and USER name to *finesaucesadmin* . Replace the *PASSWORD* value with your unique password that you used for *finesaucesadmin* user creation in the first chapter. The database is local, therefore *localhost* value for HOST.

To work with the PostgreSQL database, we will need to install *psycopg2* adapter for Python. It is available as a Python package, so let's use *pip* to get it:

```
terminal
(env) ~/finesauces$ pip install psycopg2-binary
```

2.3 Initial migration

Django applications contain a *models.py* file, where the data models are defined. Each data model is then mapped to a database table. To complete the project setup, we need to create tables associated with the models of the applications listed in *INSTALLED_APPS* in the *settings.py* file. Django includes a migration system that enables us to reflect those models into database tables. Open your terminal and run the following *migrate* command:

```
terminal
(env) ~/finesauces$ python manage.py migrate
```

If the migration process was successful, you would see the following confirmation:

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

The preceding lines are database migrations that are applied by Django. By applying migrations, tables for initial applications are created in our *finesauces* database.

You can review tables that were just created by logging into the local Postgres session. For macOS, use the Postgres app. Double click on *finesauces* database. In case you are running Linux, use the following command:

```
terminal
(env) ~/finesauces$ psql -h localhost
-d finesauces -U finesaucesadmin -p 5432
```

Once inside the session, let's list all the tables that were created for us by using *\dt* command:

```
postgres terminal
finesauces=# \dt
                  List of relations
 Schema |             Name              | Type  |  Owner
--------+-------------------------------+-------+----------
 public | auth_group                    | table | postgres
 public | auth_group_permissions        | table | postgres
 public | auth_permission               | table | postgres
 public | auth_user                     | table | postgres
 public | auth_user_groups              | table | postgres
 public | auth_user_user_permissions    | table | postgres
 public | django_admin_log              | table | postgres
 public | django_content_type           | table | postgres
 public | django_migrations             | table | postgres
 public | django_session                | table | postgres
(10 rows)
```

We can quit the PostgreSQL session by typing:

```
postgres terminal
postgres=# \q
```

You might notice that when you run Django project by using *python manage.py runserver* command again, the warning regarding 18 unapplied migrations has been resolved.

Before initializing our local repository and also pushing the project to the remote one, I want to create another *settings* file called *local_settings.py*. This file will contain our sensitive project information like database, email, and secret payment credentials. This information must always be kept confidential and never appear in the remote repository. Let's create a *local_settings.py* file in the same location as the *settings.py* file. Open both files and move *SECRET_KEY, DEBUG, ALLOWED_HOSTS,* and *DATABASES* from the *settings.py* file to the *local_settings.py* :

finesauces/finesauces_project/local_settings.py
```
# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = '<your secret key>'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []

# Database
# https://docs.djangoproject.com/en/3.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'finesauces',
        'USER': 'finesaucesadmin',
        'PASSWORD': '********',
        'HOST': 'localhost'
    }
}
```

Django framework needs these settings to run correctly; therefore, we have to import them from our *local_settings.py* file. Add the following code to the bottom of the *settings.py* file (*#... only indicates already existing code in the file – this mark itself is not part of the code*):

finesauces/finesauces_project/settings.py
```
#...
try:
     from .local_settings import *
except ImportError:
    pass
```

Let's confirm that our project is still working after this update. Run the *runserver* command in your terminal and visit http://127.0.0.1:8000/ page:

terminal
```
(env) ~/finesauces$ python manage.py runserver
```

2.4 Local repository

We initialized our Django project, tested it by running a local development server, connected to our PostgreSQL database, and performed initial migrations. Now it's time to create our local repository to keep track of our project. Let's set a local git repository by running the following command:

terminal
```
(env) ~/finesauces$ git init
```

You should receive a notification similar to this one:

terminal
```
Initialized empty Git repository in /home/peter/finesauces/.git/
```

To see a detailed log of changes done in our project since the last commit, run the *git status* command*:*

terminal
```
(env) ~/finesauces$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .vscode/
        db.sqlite3
        env/
        finesauces_project/
        manage.py

nothing added to commit but untracked files present (use "git add" to track)
```

Before committing our changes, we need to add one more file called *.gitignore* to the project. This file will contain a list of folders and files that we don't want to include in our remote repository and server, as they are redundant. Create this file in the root *finesauces* directory and add the following code to it:

finesauces/.gitignore

```
*.log
*.pot
*.pyc
__pycache__/
db.sqlite3
db.sqlite3-journal
.vscode
static
media
env
local_settings.py
```

If you rerun the *git status* command, you will see that the files and folders listed in the *.gitignore* file are now excluded:

terminal

```
(env) ~/finesauces$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        finesauces_project/
         manage.py

nothing added to commit but untracked files present (use "git add" to track)
```

Let's add our project files to the git repository by running the *git-add -A* command to keep track of them:

terminal

```
(env) ~/finesauces$ git add -A
```

Commit these files by also providing a short message describing updates in the project:

terminal

```
(env) ~/finesauces$ git commit -m "initial commit"
```

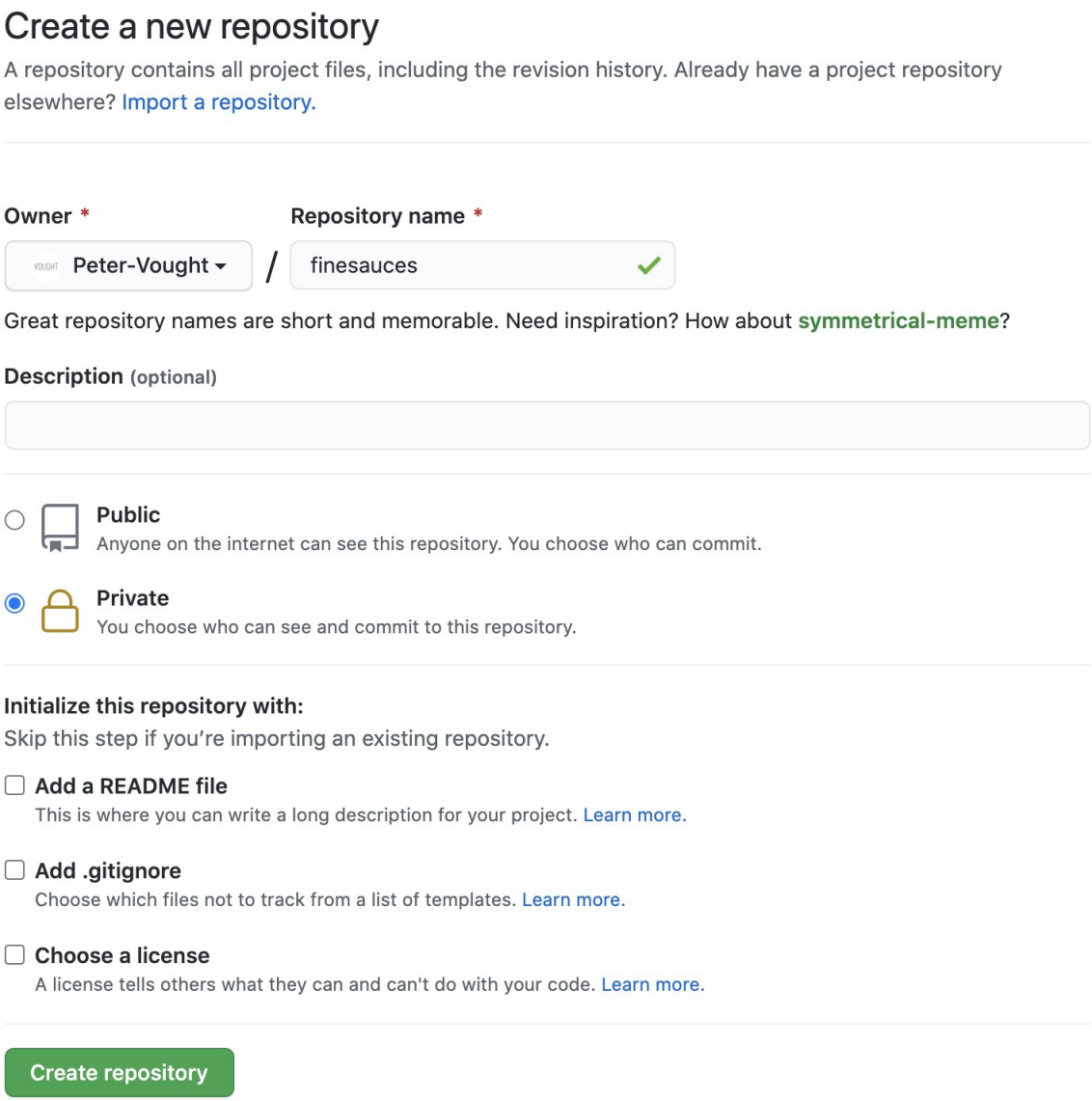
2.5 Remote repository

We just initialized our local repository. It is also recommended and will be required when deploying our project to the server to create a remote repository as well. This way, we will have a backup of our code if anything happens to our computer. It will also enable us to collaborate with other developers on the same project in the future.

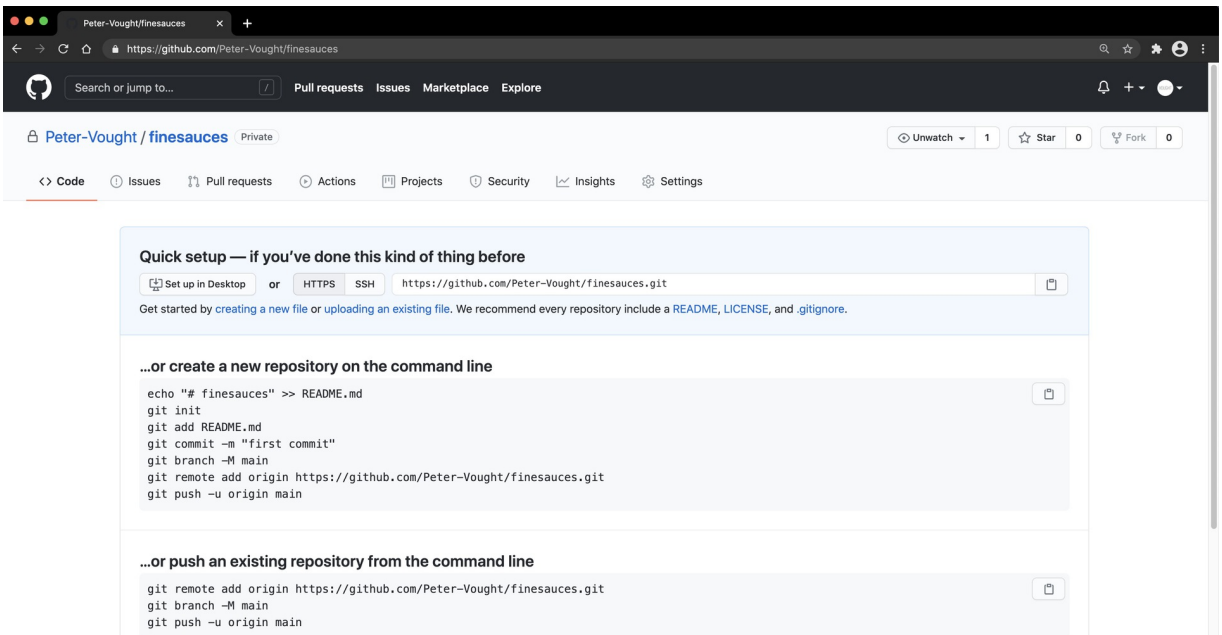
The most popular platforms for this purpose would be *Bitbucket* [8] and *Github* [9] . If you already have an account on one of these platforms, feel free to utilize that. We will use *Bitbucket* to create our repository.

Let's visit the https://github.com/ . If you haven't created your account yet, go ahead and create one at https://github.com/join_next . Fill in the required information and confirm your account.

To create our new project repository, we need to navigate to the account dashboard at https://github.com/new . Fill in the Repository name. We want to make this repository *Private* , and we don't need to include the *README* , *.gitignore* or *license* files:



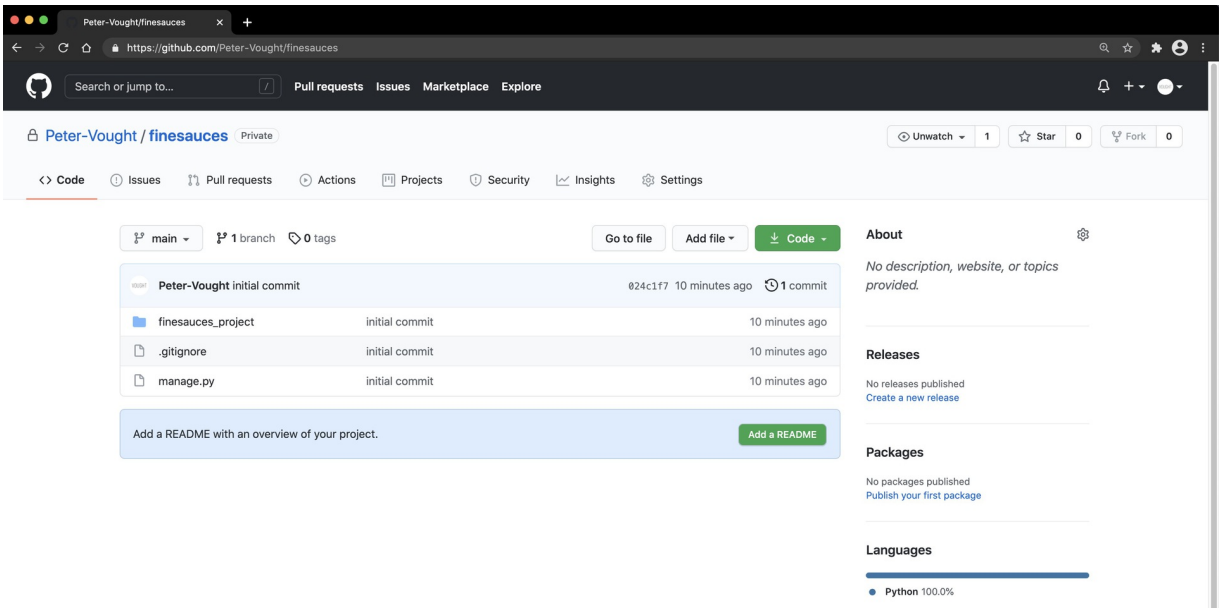
Create the repository by clicking on the *Create repository* button. A new empty repository will be created for us. Since we already have code that we want to be added here, we can focus on the *"or push an existing repository from the command line"* section:



Run the following commands from the mentioned section:

```
terminal
(env) ~/finesauces$ git remote add origin https://github.com/<username>/finesauces.git
(env) ~/finesauces$ git branch -M main
(env) ~/finesauces$ git push -u origin main
```

If you refresh the page, you will see that our local Django project is now reflected in the remote repository!


Peter-Vought/finesauces
https://github.com/Peter-Vought/finesauces
Search or jump to...
Pull requests
Issues
Marketplace
Explore
Peter-Vought / finesauces
Private
Unwatch 1
Star 0
Fork 0
Code
Issues
Pull requests
Actions
Projects
Security
Insights
Settings
main
1 branch
0 tags
Go to file
Add file
Code
Peter-Vought initial commit
024c1f7 10 minutes ago
1 commit
finesauces_project initial commit 10 minutes ago
.gitignore initial commit 10 minutes ago
manage.py initial commit 10 minutes ago
Add a README with an overview of your project.
Add a README
About
No description, website, or topics provided.
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
Languages
Python 100.0%

3 Creating listings application

Throughout this book, we will encounter the terms *project* and *application* over and over again. In Django, a project is considered a Django installation with specific settings. An application is then a group of models, views, templates, and URLs. Applications interact with the framework to provide a defined functionality. If you think of our e-commerce project, different applications would be responsible for displaying our categories and products, handling shopping cart, customer orders, and customer accounts.

Let's create our first Django application responsible for presenting our products along with corresponding categories and call it *listings*. Make sure you are within the *finesauces* directory, and your virtual environment is active. Then run the following *startapp* command in the terminal:

terminal
```
(env) ~/finesauces$ django-admin startapp listings
```

This will create the basic structure of our application, which will contain the following files and directories:

```
listings
├── __init__.py
├── admin.py
├── apps.py
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

As for this structure:

- ***admin.py*** - here we register models to include them in the Django administration site.
- ***apps.py*** - includes the main configuration of the *listings* applications.
- ***migrations*** - will contain database migrations of our application. Migrations allow Django to track model changes and synchronize the database accordingly.
- ***models.py*** - includes the data models of our application
- ***tests.py*** - we can add tests for our application inside this file.
- ***views.py*** - will contain the business logic of our application. Each view receives an HTTP request, processes it, and returns a response. This file determines how data is processed and shown to the user.

3.1 Activating listings application

Django does not know about our new application yet. For Django to keep track of our application and be able to create database tables for its models, we have to activate it. To do this, edit the *settings.py* file located in the *finesauces_project* directory and add *listings.apps.ListingsConfig* to the *INSTALLED_APPS* setting. It should look like this:

/finesauces/finesauces_project/settings.py
```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'listings.apps.ListingsConfig'
]
```

The *ListingsConfig* class defines *listings* application configuration. Now Django knows about our application and will be able to load its models.

3.2 Designing listings models

A database model is a Python class that subclasses *django.db.models.Model* [10] . It could be described as a blueprint or source of information about our data, based on which Django will generate database tables for us. Each of our models within the *models.py* file will have its very own table. Every attribute of this class will represent a database field within this table. When we create our models, Django will also provide us with a practical API to help us retrieve objects in the database [11] .

In this e-commerce project, we will be offering products belonging to a specific category. To implement this functionality, we will start by creating two models, *Category* and *Product*. Open the *models.py* file within the *listings* application and add the following code to it:

/finesauces/listings/models.py
```
from django.db import models

class Category(models.Model):
    name = models.CharField(
        max_length=100,
        unique=True
    )

    slug = models.SlugField(
        max_length=100,
        unique=True
```

```
    )

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

class Product(models.Model):
    category = models.ForeignKey(
        Category,
        related_name = 'products',
        on_delete = models.CASCADE
    )

    name = models.CharField(max_length=100, unique=True)
    slug = models.SlugField(max_length=100, unique=True)
     image = models.ImageField(upload_to='products/')
    description = models.TextField()
    shu = models.CharField(max_length=10)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    available = models.BooleanField(default=True)

    class Meta:
        ordering = ('shu',)
```

Let's go over our models and their attributes:

- ***name*** - this attribute will be responsible for preserving titles of categories and products that we create. This field is set to be a *CharField* , since we want it to store text value, which then translates into a *VARCHAR* value in our database. The maximum character count is set to 100.
- ***slug*** - This is a field intended to be used in URLs. A slug is a short label that contains only letters, numbers, underscores, or hyphens. We will use the slug field to build beautiful, SEO-friendly URLS for our e-commerce listings. Both, name and slug fields, are set to be unique, meaning Django will prevent multiple categories having the same name and slug fields.
- ***category*** - This field defines a many-to-one relationship, meaning that each product belongs to a specific category, and the category can contain any number of products. To keep track of the relationship between Category and Product models, Django will actually create extra column in the Product table to store the Category id, under which a specific product belongs. This extra category id column makes it easy to, for example, retrieve all of the products for a specific Category. As for *related_name* parameter, Django has a built-in syntax we can use, known as *FOO_set* , where *FOO* is a lowercased source model name. So, for our *Category* model, we can use *category_set* syntax to access all related *Product* instances. *related_name* parameter is entirely optional, but I prefer adding it to models, which then allows us to set the name of this reverse relationship explicitly.
- ***image*** - The image file. By default, images will be uploaded to the *products* folder.
- ***description*** - Field for storing product description. We are using *TextField* here instead of *CharField* , since we do not want to limit the description's length.
- ***shu*** - Scoville Heat Units (SHU) - since we will be selling hot sauces, we need a field to indicate, how hot the sauce is.
- ***price*** - this field uses Python's *decimal.Decimal* type to store a fixed precision decimal number. The maximum number of digits (including the decimal places) is set using the *max_digits* attribute and decimal places with *decimal_places* attribute. We use *DecimalField* instead of *FloatField* to avoid rounding issues.
- ***available*** - A Boolean value to indicate whether the product is available or not. If no value *(True/False)* is specified, a database record is created with the default value specified *(True)*.

To provide any additional information or metadata about our models, we use inner class *Meta*. As per Django documentation [12] , model metadata is *"anything that's not a field"*. Note that adding *Meta* class to our models is optional. You can find the list of available Meta options in the official Django documentation [13] .

In the *Meta* class of both *Category* and *Product* , we tell Django to sort the results by the *name* field in descending order for Category model and by *shu* field for Product model by default when we query the database.

Since we are going to deal with images in our models, we also need to install the *Pillow* library by using the following command:

```
terminal
(env) ~/finesauces$ pip install Pillow
```

3.3 Creating and applying migrations

In the previous section, we successfully defined our *Category* and *Product* models. To have them reflected in the database tables, we need to run the *makemigrations* command, which is essentially responsible for keeping track of our model changes. All of those changes are tracked within individual migration files.

```
terminal
(env) ~/finesauces$ python manage.py makemigrations
```

After executing this command, Django provides us with the following output:

```
terminal
Migrations for 'listings':
  listings/migrations/0001_initial.py
    - Create model Category
    - Create model Product
```

Our first migration file, called *0001_initial.py,* was generated within the *migrations* folder in our *listings* application. To see the actual SQL code that will be executed for specified migration, we can run the *sqlmigrate* [14] command. Run the following command to inspect the SQL for our first migration:

```
terminal
(env) ~/finesauces$ python manage.py sqlmigrate listings 0001
```

You will see the following output in your terminal console:

```
terminal
BEGIN;
--
-- Create model Category
--
CREATE TABLE "listings_category" ("id" serial NOT NULL PRIMARY KEY, "name"
varchar(100) NOT NULL UNIQUE, "slug" varchar(100) NOT NULL UNIQUE);
--
-- Create model Product
--
CREATE TABLE "listings_product" ("id" serial NOT NULL PRIMARY KEY, "name"
varchar(100) NOT NULL UNIQUE, "slug" varchar(100) NOT NULL UNIQUE, "image"
varchar(100) NOT NULL, "description" text NOT NULL, "shu" varchar(10) NOT
NULL, "price" numeric(10, 2) NOT NULL, "available" boolean NOT NULL,
"category_id" integer NOT NULL);
CREATE INDEX "listings_category_name_6b0e34ae_like" ON "listings_category"
("name" varchar_pattern_ops);
CREATE INDEX "listings_category_slug_2977b272_like" ON "listings_category"
("slug" varchar_pattern_ops);
ALTER TABLE "listings_product" ADD CONSTRAINT
"listings_product_category_id_5d2fa1ec_fk_listings_category_id" FOREIGN KEY
("category_id") REFERENCES "listings_category" ("id") DEFERRABLE INITIALLY
DEFERRED;
CREATE INDEX "listings_product_name_c40fa5c8_like" ON "listings_product"
("name" varchar_pattern_ops);
CREATE INDEX "listings_product_slug_165302a8_like" ON "listings_product"
("slug" varchar_pattern_ops);
CREATE INDEX "listings_product_category_id_5d2fa1ec" ON "listings_product"
("category_id");
COMMIT;
```

The exact SQL code might vary based on the database system used. Django generates the table names by combining the application name and the lowercase name of the model (*listings_category, listings_product*).

Run the *migrate* command to synchronize the database with the current set of models and migrations:

```
terminal
(env) ~/finesauces$ python manage.py migrate
```

After successful migration, we get the following response:

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

After applying the migrations, the database reflects the current status of our models. We can confirm that by visiting our *listings* database in Postgres app:

Once logged in, use the following command to list all the tables within *listings* database:

```
postgres terminal
finesauces=# \dt
```

You can notice additional two tables that were created for our *Category* and *Product* models:

```
postgres terminal
                List of relations
 Schema |            Name             | Type  |  Owner
--------+-----------------------------+-------+----------
 #...
 public | listings_category           | table | postgres
 public | listings_product            | table | postgres
(12 rows)
```

Remember that after each update to the *models.py* file in order to add, remove, or change the fields of existing models or to add new models, you will have to create a new migration using the *makemigrations* command. The migration will allow Django to keep track of model changes. You will then have to apply it with the *migrate* command to keep the database up to date with your models.

3.4 Creating administration site

Now that we have defined *Category* and *Product* models, we will utilize Django's administration site to create a couple of records for our e-commerce project. Django comes with a built-in administration interface built dynamically by reading our model data and providing a production-ready interface for managing content. If you, let's say, create a Django project for your client, it will enable them to update content without touching any code. Saying that the Django admin site is a great feature would be a significant understatement. Not many web frameworks offer or can compare to such out-of-the-box functionality.

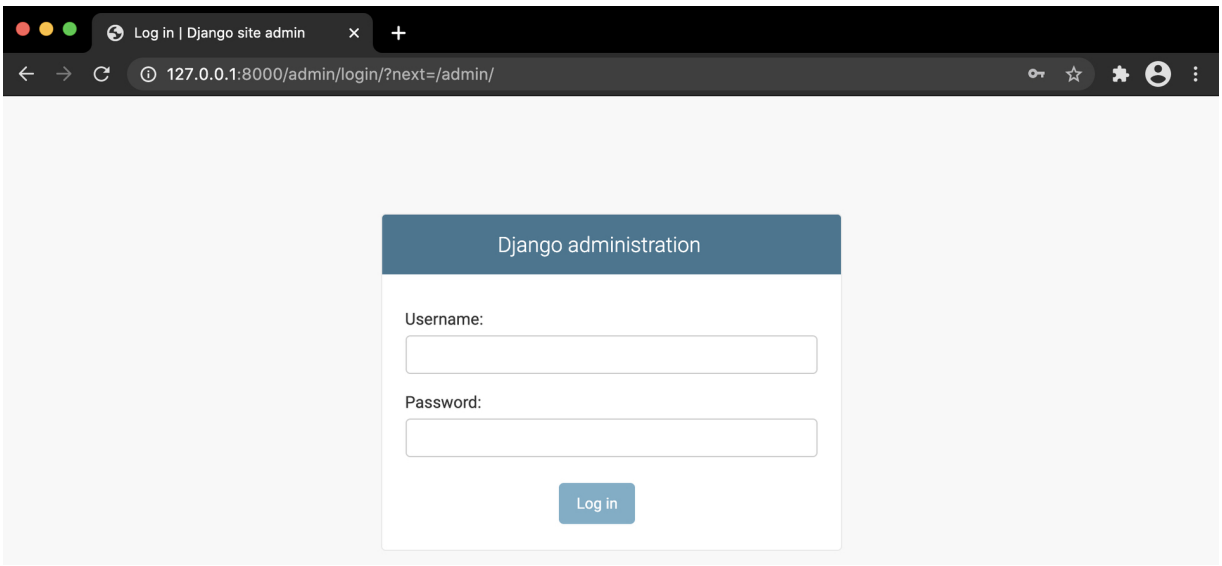
3.4.1 Creating superuser

To access and manage the administration site, we will need to create a superuser first. Run the following command in your terminal and proceed through the prompts:

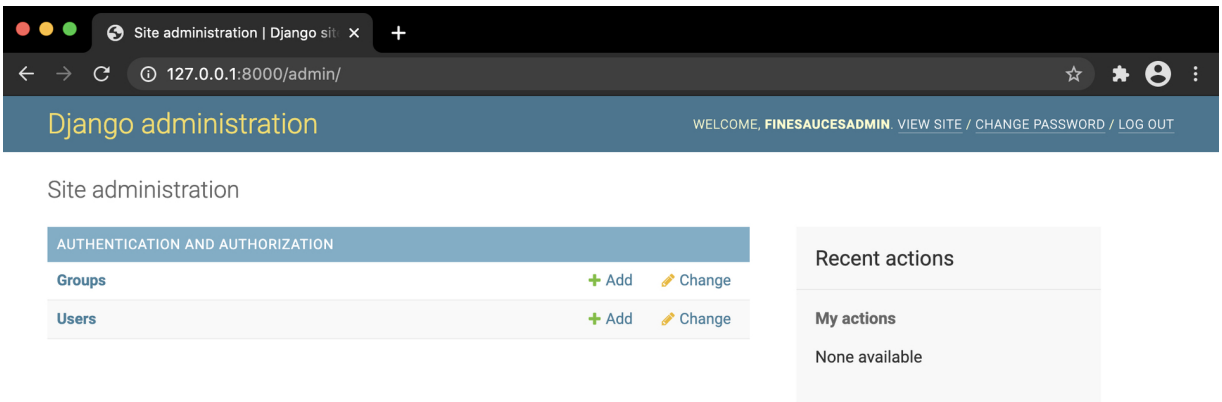
```
terminal
(env) ~/finesauces$ python manage.py createsuperuser
Username (leave blank to use 'peter'): finesaucesadmin
Email address: admin@finesauces.store
Password:
Password (again):
Superuser created successfully.
```

3.4.2 Accessing administration site

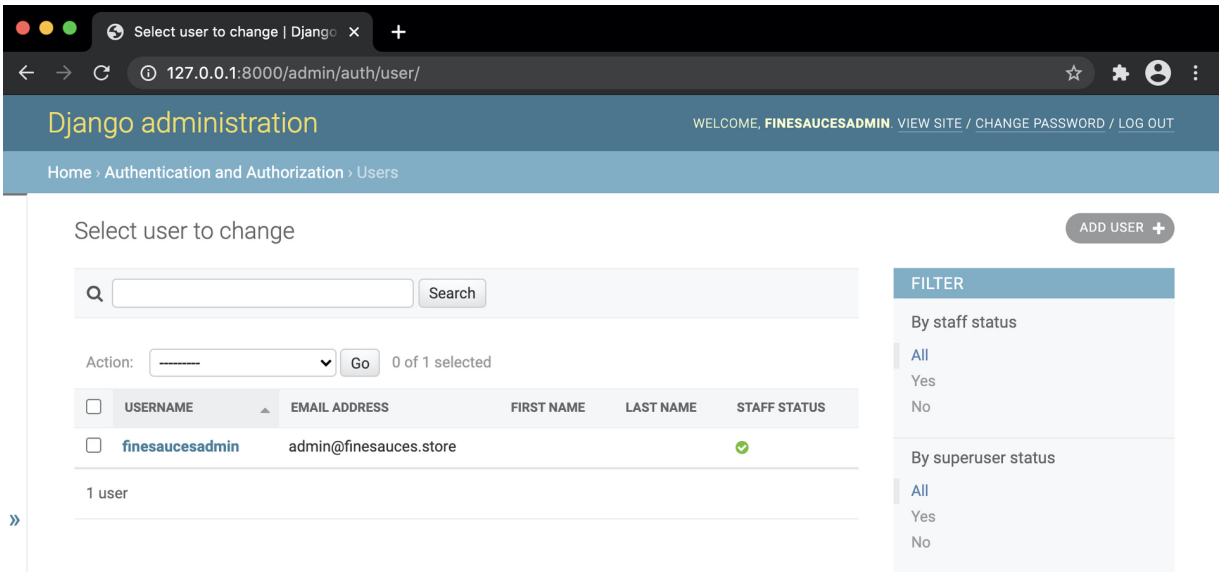
Restart your local development server with the *python manage.py runserver* command and access the admin login page http://127.0.0.1:8000/admin/



Log in using the superuser credentials we just created. You will see the following administration site index page:



The *Groups* and *Users* models are part of the Django authentication framework located in *django.contrib.auth* [15] . If you click on *Users* , you will see the user you created previously.



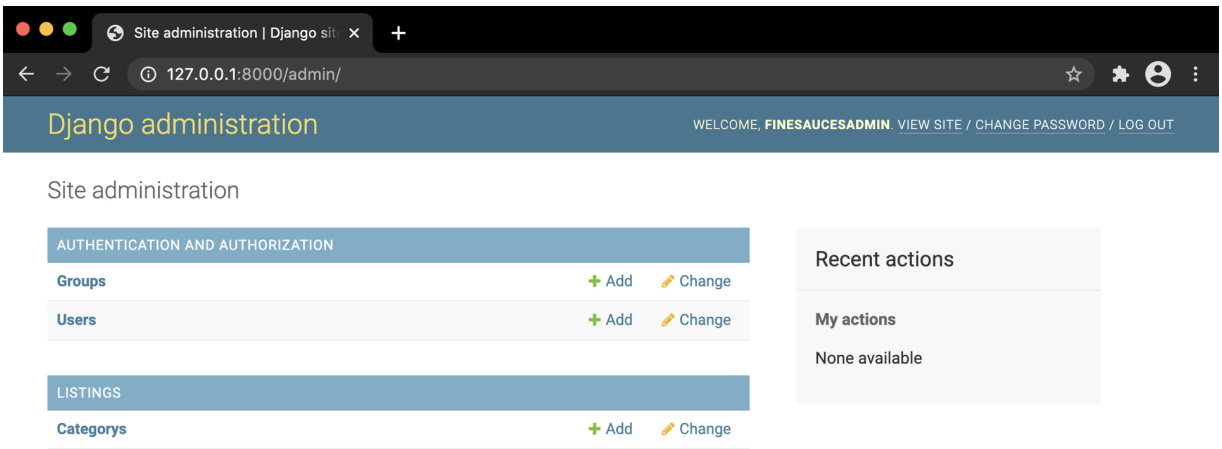
3.4.3 Adding models to the administration site

For now, we are not able to see our custom *Category* and *Product* models on the admin site. We need to modify *admin.py* file residing in our *listings* application. Open the file and update it to look this:

/finesauces/listings/admin.py

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

admin.site.register(Category)
```

If we now return back to the landing admin site http://127.0.0.1:8000/admin/ , we will see our *Category* model available:



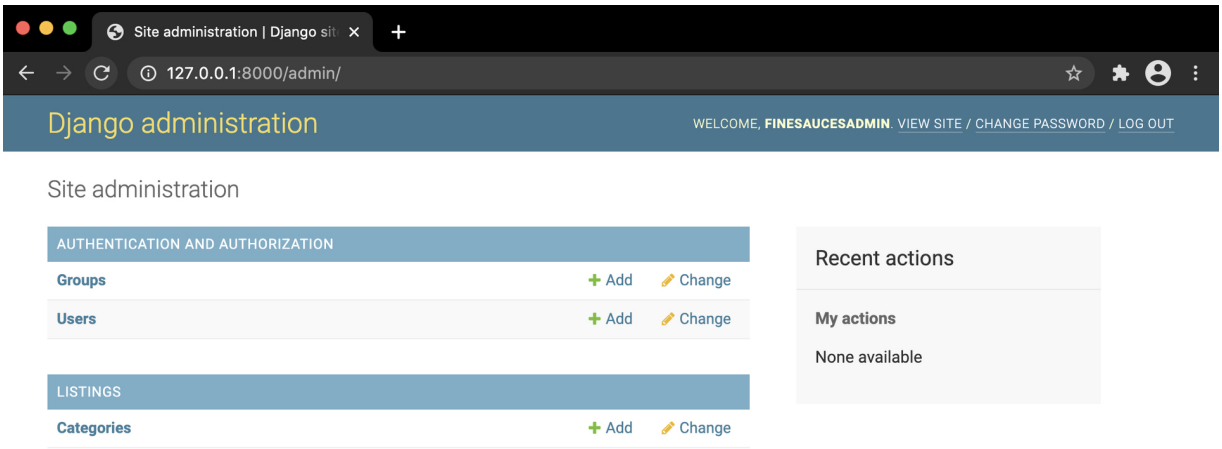
When we register the model in the Django administration site, we get a user-friendly interface generated by introspecting our models what allows us to list, edit, create, and delete objects in a simple way.

However, you have probably noticed a little issue with our *Category* model. Django, by default, modifies the class name by appending *s* to the end. *Category* , therefore, becomes *Categorys* , which might result in slight naming inconsistencies. Lucky for us, it's easy to override this naming formula by updating our *models.py* file of *listings* application. Find the *Meta* class inside the *Category* model and update it to include *verbose_name_plural* [16] attribute:

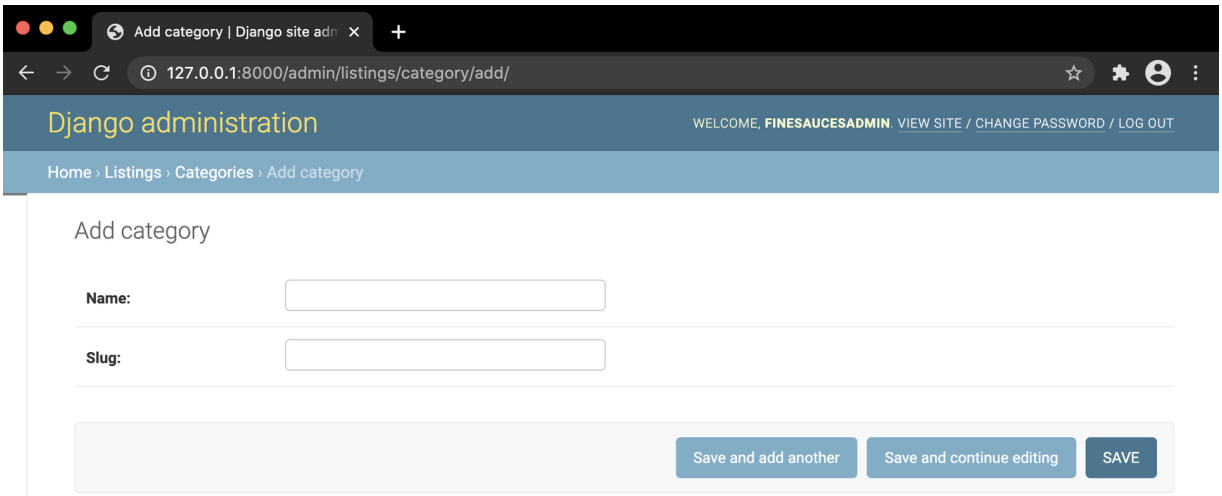
/finesauces/listings/models.py

```
#...
class Meta:
    ordering = ('-name',)
    verbose_name_plural = 'categories'
    #...
```

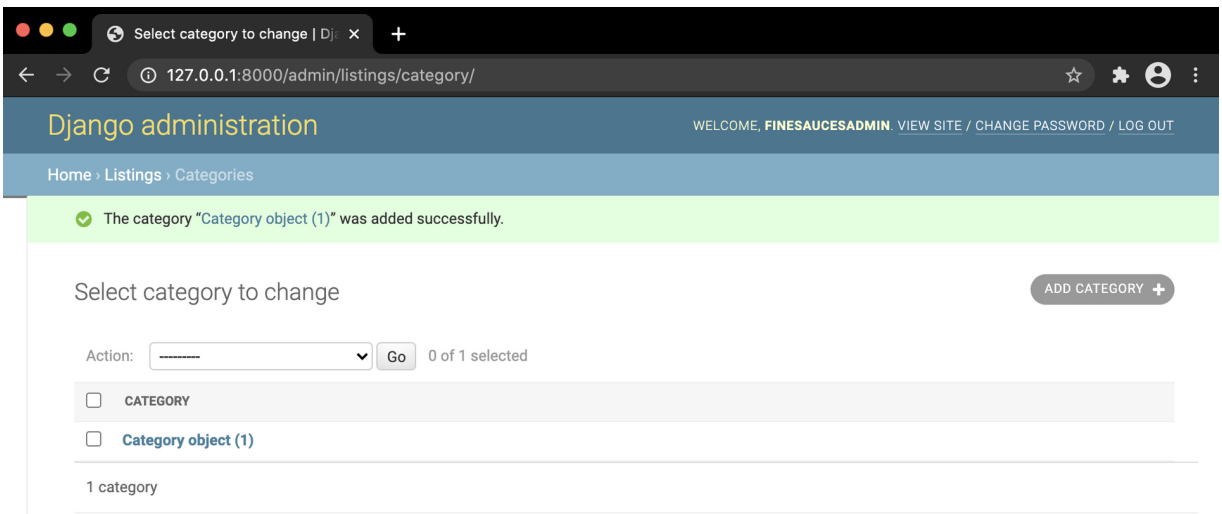
When you reload the admin site, you will notice the naming adjusted appropriately:



Click on the *Add* link besides *Categories* to add a new *Category* . You will be presented with the form generated dynamically by Django for our model, as shown in the following screenshot:



Go ahead and fill in the new category's name (since we are in the hot sauce business now, I decided to go with the *Mild*) and slug field *(mild)* and click on *SAVE* button. You should be redirected back to the *Categories* list page with a success message and the category you just created. However, if you look closely, there's a problem: our new entry is called *"Category object"* , which isn't very helpful.

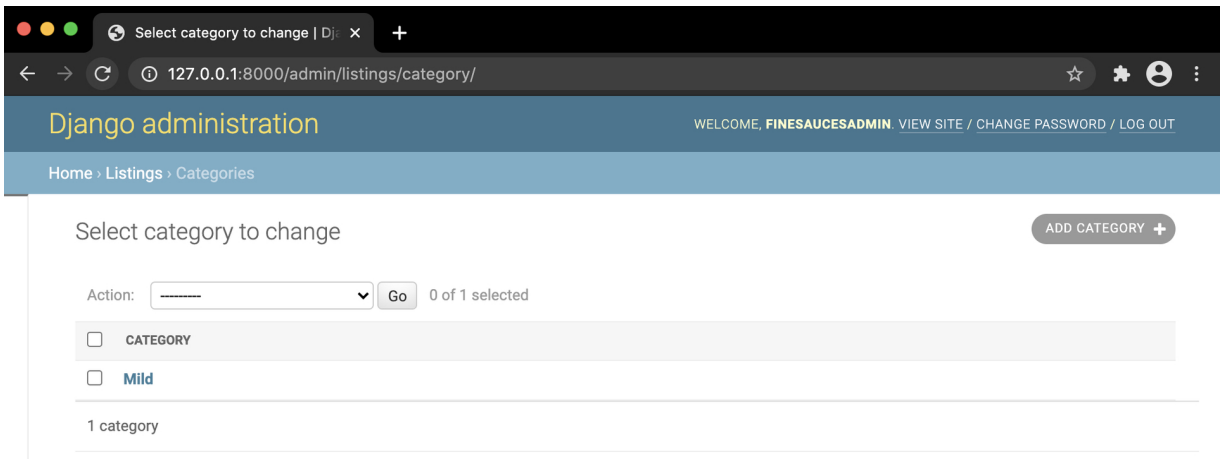


It would be much better to see the actual name of our category. Return back to the *Category* model inside *models.py* file and add the *__str__* function as follows:

```
/finesauces/listings/models.py
#...
class Category(models.Model):
    name = models.CharField(
        max_length=100,
        unique=True
    )

    slug = models.SlugField(
        max_length=100,
        unique=True
    )

    class Meta:
        ordering = ('-name',)
        verbose_name_plural = 'categories'
```

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

Add the *__str__* function to *Product* model in the same manner. If you refresh your Admin page in the browser, you'll see it changed to a much more descriptive and helpful representation of our database entry.



Note that you can format the string returned from this function according to your needs.

3.4.4 Customizing how models are displayed

In the previous section, we managed to modify our admin panel to display *Category* model. Let's push it a bit further and adjust the way our model is displayed. Edit the *admin.py* file of your *listings* application and change it to:

```
/finesauces/listings/admin.py
from django.contrib import admin
from .models import Category

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

We have replaced *admin.site.register(Category)* with a decorator to register our *ModelAdmin* class. By defining *the list_display option, we tell* Django to show specific fields in the admin area, *name*, and *slug* in our case. *list_display* could be also replaced by *fields* option *(e.g. fields = ('name', 'slug'))*. Unlike *list_display* , however, *fields* attribute does not accept *callables* , custom functions that you might define to customize the admin panel even further (we will explore this later in the *Extending administration site* chapter).

prepopulated_fields attribute is a dictionary mapping field names to the fields it should prepopulate from. The value is automatically set using the value of other fields, which is convenient, especially for generating slug fields. Given field utilizes a bit of Javascript to populate the assigned field. As per Django documentation [17] , *"generated value is produced by concatenating the values of the source fields, and then by transforming that result into a valid slug (e.g. substituting dashes for spaces; lowercasing ASCII letters; and removing various English stop words such as 'a', 'an', 'as', and similar)"*.

Let's finish this section by adding *Product* model to the admin site as well. Add the following code to the *admin.py* file of *listings* application:

```
/finesauces/listings/admin.py
from django.contrib import admin
from .models import Category, Product

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

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

We added a couple of new attributes here. *list_filter* will enable us to filter products by their category and availability. When we create some products, you will notice a sidebar in the *Products* section very similar to the one in *Users* . We will use the *list_editable* attribute to set the fields that can be edited from the administration site's display page. This will allow us to edit multiple rows at once. Any field in *list_editable* must also be present in the *list_display* attribute since only the displayed fields can be edited.

While we are in the admin site, let's add one more category to the portfolio (I will go with *Extreme* . Notice how the *slug* field is being filled in automatically).

Before creating any *Product* records, we will have to update *settings.py* and *urls.py* files in our *finesauces_project* directory. Since we are using *ImageField* to store product images, we need the development server to serve uploaded image files.

Edit the *settings.py* file of *finesauces_project* and add the following settings:

/finesauces/finesauces_project/settings.py
```
from pathlib import Path
import os

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
#...
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.1/howto/static-files/

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

try:
    from .local_settings import *
except ImportError:
    pass
```

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

For Django to serve the uploaded media files using the development server, edit the main *urls.py* file of *finesauces_project* and modify it to look like this:

/finesauces/finesauces_project/urls.py
```
from django.contrib import admin
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
]
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

We are serving static files this way only during the development phase. You should never serve static files with Django in a production environment, as the Django development server does not serve them efficiently.

Let's create a couple of products now. When creating a new product, you can notice that you can directly select any category from the dropdown menu and that the slug field for the product name is being generated automatically (*for generating dummy text, I like to use Cupcake Ipsum generator* [18] *, that gives you little more entertaining text to work with*).

3.5 Displaying our categories and products

In the previous section, we have created a couple of categories and products for our e-commerce project. Let's work on displaying these records now. We will start by building our first view. A Django view is just a Python function that receives a web request and returns a web response. This response can be the HTML contents of a Web page, redirect, 404 error, an XML document, or an image. The view contains the business logic. It defines what is shown to the user and returns the appropriate response. The convention is to put the views inside *views.py* file in your project or application directory [19] .

3.5.1 Building our product list view

Let's start by creating our view to display a list of products we previously created in the admin site. Edit the *views.py* file of our *listings* application and make it look like this:

/finesauces/listings/views.py
```
from django.shortcuts import render
from .models import Category, Product

def product_list(request):
    categories = Category.objects.all()
    products = Product.objects.all()

    return render(
        request,
        'product/list.html',
        {
            'categories': categories,
            'products': products
        }
    )
```

We just created our first Django view. The *product_list* view takes the *request* object as the only parameter. This parameter is required by all views. Then we retrieve the collections of *Category* and *Project* objects(called *QuerySets* [20]) available in our database. We use the *render* function to render retrieved collections within *list.html* template (which we will create in the next section). The *render()* shortcut also includes optional third argument, dictionary (called *context*), containing our objects, which are then available for the template to use.

3.5.2 Creating template for our product list

Templates represent a convenient way to generate HTML files and present requested data to the user. A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted. To put is simply, templates define how data is displayed. They are usually written in HTML in combination with Django template language.

So where do we put our templates? Django by default looks for *templates* folder within each application. Let's go ahead and create this folder within our *listings* application. Then, within *templates* folder, create *base.html* file, *product* folder and within it create *detail.html* and *list.html* files. The final structure of the *templates* folder should look like this:

```
templates/
├── base.html
└── product
    ├── detail.html
    └── list.html
```

This will be our basic templates structure for *listings* application. The *base.html* file will include the main HTML structure of our page (navbar). *list.html* and *detail.html* will extend this page to render categories and products provided by our *product_list* view. The idea behind *base.html* file is to include the code that will be common to all pages and prevent repeating it in every template (great example would be a page navbar).

Django also comes with its own template system called Django template language (DTL [21]), which enables us to customize, how the template is presented. It comes with four constructs:

- ***variables*** - allow us to render a value from the *context* . They are surrounded by **{{** and **}}**
- ***tags*** - provide arbitrary logic in the rendering process. They most often serve as a control structure in shape of *"if"* statement or a *"for"* loop. Tags are surrounded by **{%** and **%}**
- ***filters*** - transform the values of variables and tag arguments. They are surrounded by **{{** and **}}** and come after variable we want to transform, separated by | (e.g. {{ review.created**|date** }}). They can also take arguments: {{ review.created**|date:"Y-m-d"** }}
- ***comments*** - look like this: *{# this won't be rendered #}*. A **{% comment %}** tag provides multi-line comments and has to be concluded by {% endcomment %}

We define *{% block title %}* and *{% block content %}* tags in *base.html* template, which will be utilized for extending this template.

/finesauces/listings/templates/base.html

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
            <link      rel="stylesheet"      href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.13.0/css/all.min.css">
                                                  <link                          rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
    integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
    crossorigin="anonymous">
  <link rel="stylesheet" href="https://fonts.googleapis.com/css2?family=Nunito&display=swap">
  <link rel="stylesheet" href="{% static 'css/style.css' %}">
  <title>{% block title %}{% endblock %}</title>
</head>
<body>
  <nav class="navbar navbar-expand-md navbar-dark bg-danger fixed-top py-1"
    id="main-nav">
    <div class="container">
      <a href="" class="navbar-brand">
        <h3 class="font-weight-bold">finesauces</h3>
      </a>
    </div>
  </nav>

  <div class="container py-5">
  {% block content %}
  {% endblock content %}
  </div>
  <script src="https://code.jquery.com/jquery-3.5.1.min.js"
    integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
    crossorigin="anonymous"></script>
  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
    integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
    crossorigin="anonymous"></script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/js/bootstrap.min.js"
    integrity="sha384-smHYKdLADwkXOn1EmN1qk/HfnUcbVRZyYmZ4qpPea6sjB/pTJ0euyQp0Mk8ck+5T"
    crossorigin="anonymous"></script>
   <script src=" {% static 'js/scripts.js' %}"></script>
   <script  src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js">
</script>
</body>
</html>
```

{% load_static %} tells Django to load the static template tags that are provided by the *django.contrib.staticfiles* application, which is included in the *INSTALLED_APPS* setting. After loading these tags, we are able to use *{% static %}* template tag

throughout this template. With this template tag, we can include the static files, which define our application styling and functionality (via javascript), such as the *style.css* file.

To obtain custom *css* styling and *javascript* code for this project, copy the contents of *static/* directory located in the *listings* application of complete application available for download at https://github.com/Peter-Vought/finesauces/tree/main/listings/static into your project. Do not copy admin.css file for now.

Now let's edit the *list.html* file and make it look like the following:

/finesauces/listings/templates/product/list.html

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

{% block title %}finesauces{% endblock %}

{% block content %}
<div class="row justify-content-center">
  <div class="col-lg-3">
    <div class="card shadow-custom border-0">
      <h4 class="p-3 font-weight-bold text-grey">
        Categories
      </h4>
      <ul class="list-group border-0 categories_list">
        <li class="list-group-item border-0 p-0 selected">
          <a href="" class="btn btn-block text-left">All</a>
        </li>
        {% for category in categories %}
          <li class="list-group-item border-0 p-0 unselected">
             <a href="" class="btn btn-block text-left">
              {{ category.name }}
            </a>
          </li>
        {% endfor %}
      </ul>
    </div>
  </div>
  <div class="col-lg-6">
    <div class="card shadow-custom border-0 mb-4">
      <div class="card-body text-center">
        <h2 class="font-weight-bold text-grey">
          Our products
        </h2>
        <div class="bottom_line"></div>
      </div>
    </div>
    <div class="row no-gutters">
    {% for product in products %}
      <div class="col-sm-6">
        <div class="card product">
          <div class="item">
            <div class="item-image">
              <img src="{{ product.image.url }}" alt=""
                class="img-fluid rounded-0">
            </div>
            <div class="item-text">
              <div class="item-text-wrap">
                <a href="" class="h2 text-white text-decoration-none">
                  <i class="fas fa-search-plus"></i>
                </a>
              </div>
            </div>
            <div class="card-body text-center">
              {{ product.name }}
              <hr>
              <span>${{ product.price }}</span>
              <div>
                <i class="fas fa-fire"></i> {{ product.shu }} SHU
              </div>
            </div>
          </div>
         </div>
      </div>
    {% endfor %}
    </div>
  </div>
  <div class="col-lg-3"></div>
</div>
<hr>
{% endblock content %}
```

With the *{% extends %}* template tag, we tell Django to inherit from the *base.html* template. Then, we fill the *title* and *content* blocks of the base template with our content.

In the *list.html* file, we utilize the *Django template language* "*for*" loop to list all categories and products at our disposal. Remember that *categories* and *products* variables come from our *product_list* view as the optional dictionary (context) argument.

For example, to access our variable attributes, the name of the category (*category.name*), we use dot-lookup syntax. We can access all of the attributes specified in your *models.py* file this way.

3.5.3 Adding URL pattern for our view

As we already established, Django *views* determine *what* is being shown to the user. *URLconf* then determines *where* the content is shown. *URLconf* is a Python code representing a mapping between URL path expressions and our Python functions (views) [22] . When the user tries to access our landing page, Django loads our URL module and looks for the variable *urlpatterns* **.** Django then goes through each URL pattern from top to bottom and stops at the first one that matches the requested URL. Then, a view of the matching URL pattern is imported and executed while being provided an instance of the *HttpRequest* class and the keyword or positional arguments.

Create a *urls.py* file in the directory of the *listings* application and add the following lines to it:

/finesauces/listings/urls.py
```
from django.urls import path
from . import views

app_name = 'listings'

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

We start by importing *path* and our listings *views* . We also defined an application namespace with the *app_name* variable. This allows us to organize URLs by application and use the name when referring to them.

Now we have to include the URL patterns of the *listings* application in the project's main URL patterns. Edit the *urls.py* file located in the *finesauces_project* directory and modify it to look like this:

/finesauces/finesauces_project/urls.py
```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('listings.urls', namespace='listings')),
]
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

The *include()* function allows referencing other URLconfs. Whenever Django encounters *include()*, it removes whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing. As per Django documentation [23] , "*You should always use* ***include()*** *when you include other URL patterns.* ***admin.site.urls*** *is the only exception to this.*"

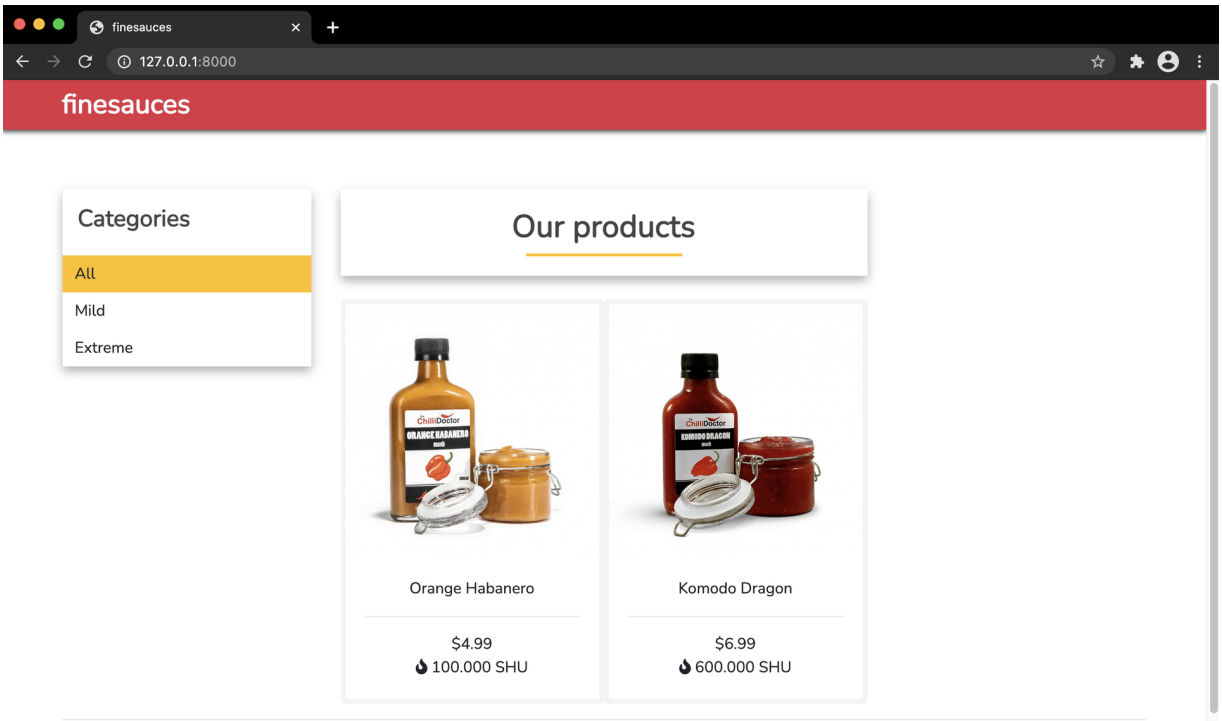
The new URL ' ' refers to the patterns defined in the *listings* application and will serve as our landing page. Let's add this URL to the navbar located in *base.html* file. Find the following code:

/finesauces/listings/templates/base.html
```
<div class="container">
  <a href="" class="navbar-brand">
    <h3 class="font-weight-bold">finesauces</h3>
  </a>
</div>
```

and modify *href* to include our new URL:

/finesauces/listings/templates/base.html
```
<div class="container">
  <a href="{% url 'listings:product_list' %} " class="navbar-brand">
    <h3 class="font-weight-bold">finesauces</h3>
  </a>
</div>
```

Start your development server by using *python manage.py runserver* command and open https:127.0.0.1:8000/ in your browser to see our new landing page:



3.5.4 Filtering by category

Now that we are able to show our categories and products on landing page of our ecommerce project, it would be great idea to filter our products based on the category they belong to. We will still be using the same *list.html* template and the same *product_list* view. All it takes is a little tweaking.

Let's start by modifying *views.py* file to look like this:

/finesauces/listings/views.py

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

def product_list(request, category_slug=None):
    categories = Category.objects.all()
    requested_category = None
    products = Product.objects.all()

    if category_slug:
        requested_category = get_object_or_404(Category, slug=category_slug)
        products = Product.objects.filter(category=requested_category)

    return render(
         request,
        'product/list.html',
        {
            'categories': categories,
            'requested_category': requested_category,
            'products': products
        }
    )
```

We are importing *get_object_or_404* shortcut to retrieve the desired category if the view is provided category slug. In case no category is found, this shortcut will throw a 404 error. Django will catch it and return the standard error page for our application, along with the HTTP error code 404 [24] . We will be providing *category_slug* to the view when we click on one of our categories on the landing page. In that case, the *product_list* view proceeds to fetch the corresponding category and all of the products for that category. We then pass it to the render function to have it available in our template. To pass our *category_slug* to product_list view, we will need to modify *urls.py* file residing in *listings* application to include another URL that will be responsible for passing slug to our view:

/finesauces/listings/urls.py

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

app_name = 'listings'

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

We have now defined two URL patterns for the *product_list* view. A pattern named *product_list* , which calls the *product_list* view without any parameters, and a pattern named *product_list_by_category* , which provides a *category_slug* parameter to our view.

Our latest URL containing category slug will be, of course, unique to each category. Or, in other words, each category will have a unique URL pointing to it. We will follow Django's suggestion here and add *get_absolute_url()* [25] in our model. This is a best practice that you should always follow. It sets a canonical URL for an object, so even if your URLs' structure changes in the future, the reference to the specific object is still the same.

Open *models.py* file of *listings* application and make the following additions to it:

/finesauces/listings/models.py
```
from django.db import models
from django.urls import reverse

class Category(models.Model):
    #...
    def get_absolute_url(self):
        return reverse(
            'listings:product_list_by_category',
            args=[self.slug]
        )
```

We import *reverse()* method, which allows us to build URLs by their name and pass optional parameters, in our case category *slug* , which is passed as an argument in our *listings:product_list_by_category* URL, that we reference in *get_absolute_url()* function.

Let's wrap up this section by modifying *list.html* template to take advantage of new *product_list* view and urls. Find the following code snippet in the template:

/finesauces/listings/templates/product/list.html
```
<li class="list-group-item border-0 p-0 selected">
  <a href="" class="btn btn-block text-left">All</a>
</li>
```

and modify it to:

/finesauces/listings/templates/product/list.html
```
<li
  {% if not requested_category %}
    class="list-group-item border-0 p-0 selected"
  {% else %}
    class="list-group-item border-0 p-0 unselected"
  {% endif %}>
    <a href="{% url 'listings:product_list' %}"
      class="btn btn-block text-left">All</a>
</li>
```

Whole point of this update is to check, whether any category was selected, and if not, highlight the *All* field in the selection menu by using *selected* styling class. We also add the URL to point to landing page.

Now we will implement very similar change to the categories list. Find the following code:

/finesauces/listings/templates/listings/list.html
```
{% for category in categories %}
  <li class="list-group-item border-0 p-0 unselected">
    <a href="" class="btn btn-block text-left">
      {{ category.name }}
    </a>
  </li>
{% endfor %}
```

and update it to look like this:

/finesauces/listings/templates/listings/list.html
```
{% for category in categories %}
  <li
    {% if category.slug == requested_category.slug %}
      class="list-group-item border-0 p-0 selected"
    {% else %}
      class="list-group-item border-0 p-0 unselected"
    {% endif %}>
       <a href="{{ category.get_absolute_url }}"
        class="btn btn-block text-left">
        {{ category.name }}
      </a>
  </li>
{% endfor %}
```

If the slug of the category we requested equals to the slug of the category we are iterating through, we mark it as selected and highlight it. We also *utilize get_absolute_url* functionality to build URLs for all of the categories. When we visit a specific category, it would be reasonable to replace the header *Our Products* with the category's name. Find the following code:

/finesauces/listings/templates/list.html
```
<h2 class="font-weight-bold text-grey">
  Our products
```

```
</h2>
```

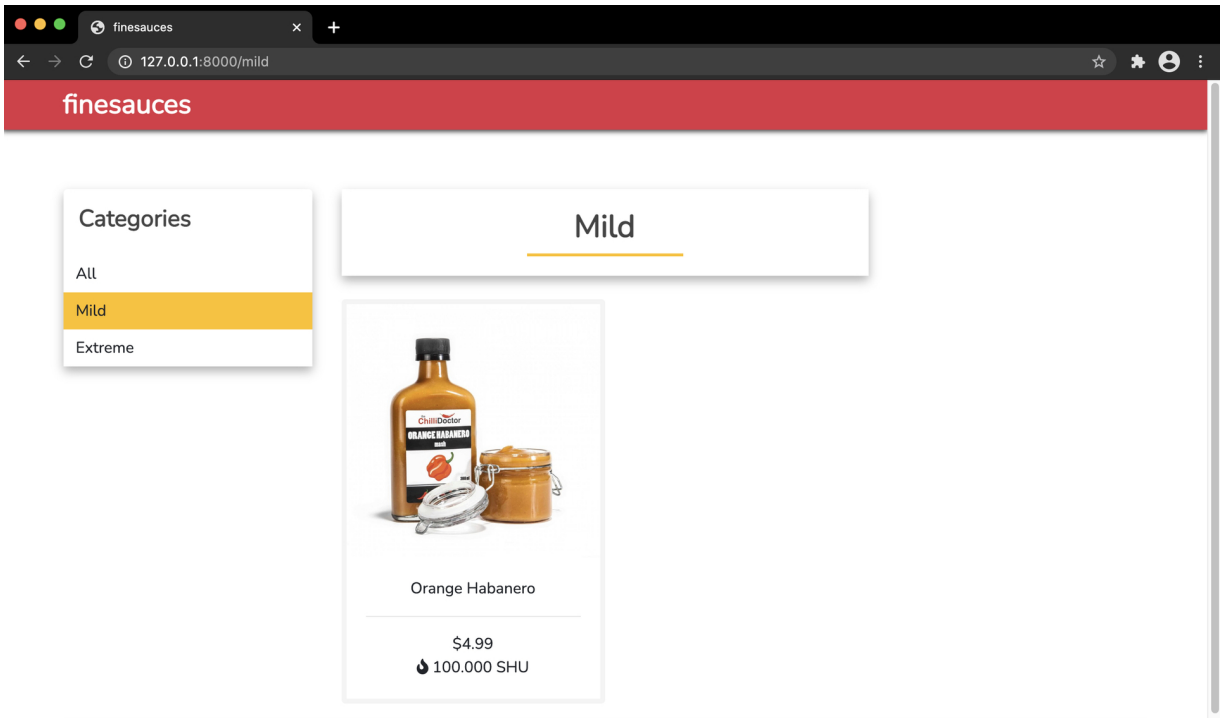
and update it to:

/finesauces/listings/templates/list.html

```
<h2 class="font-weight-bold text-grey">
  {% if requested_category %}
    {{ requested_category.name }}
  {% else %}
     Our products
  {% endif %}
</h2>
```

In case a specific category is requested, we print its name instead of *Our products* .

Feel free to access any of the categories from the list on the landing page now. You will be presented with the filtered version of the page corresponding to the category you selected:



3.5.5 Product detail page

Now that we are able to filter our products based on the category, we can work on displaying detailed page for our products, so users can read a little more about hot sauces that we sell. To implement this, we will start by modifying *views.py* and creating a new view:

/finesauces/listings/views.py

```
def product_detail(request, category_slug, product_slug):
    category = get_object_or_404(Category, slug=category_slug)
    product = get_object_or_404(
        Product,
        category_id = category.id,
        slug=product_slug
    )

    return render(
        request,
        'product/detail.html',
        {
            'product': product
        }
    )
```

This view is pretty straightforward. We provide *category slug* and *product slug* and fetch the corresponding product. We then pass it to our *detail.html* template, which we will build shortly. You might wonder where does *category_id* field comes from. Django, by default, adds it as the last column to our table to keep track of the relationship between category and product we defined in our *models.py* file.

Now, modify *urls.py* file inside your *listings* application to add new URL, that will help us retrieve a specific product. The updated file should look as follows:

/finesauces/listings/urls.py

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

app_name = 'listings'

urlpatterns = [
```

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

Let's modify *models.py* file of *listings* application to include canonical URL for our *Product* model. Add the *get_absolute_url()* function as follows:

/finesauces/listings/models.py
```
Class Product(models.Model):
    #...
    def get_absolute_url(self):
        return reverse(
            'listings:product_detail',
            args=[self.category.slug, self.slug]
        )
```

While we are at it, we can update the product list in *list.html* template so that each product card contains URL to its product detail page. Find the following section within *{% for product in products %}* loop:

/finesauces/listings/templates/product/list.html
```
<a href="" class="h2 text-white text-decoration-none">
  <i class="fas fa-search-plus"></i>
</a>
```

and modify the *href* to include product canonical URL:

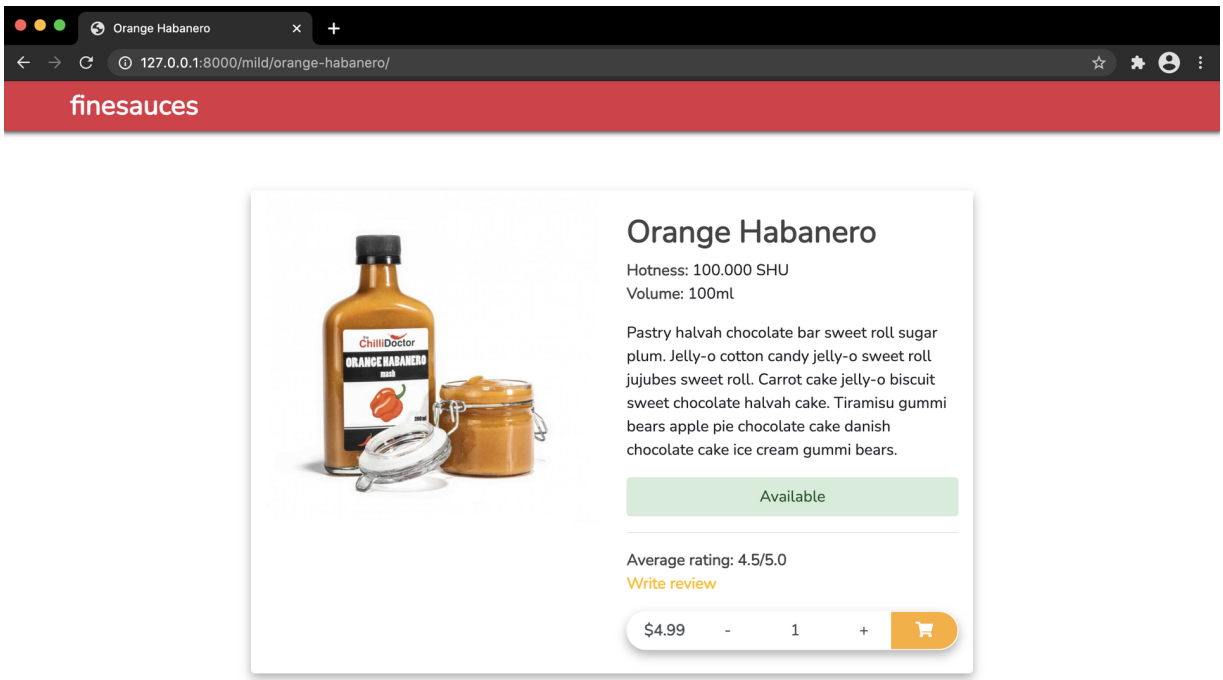
/finesauces/listings/templates/product/list.html
```
<a href="{{ product.get_absolute_url }} "
  class="h2 text-white text-decoration-none">
  <i class="fas fa-search-plus"></i>
</a>
```

Now that our view and URLs are ready, last piece we need to build is product detail template. Open *detail.html* file within *templates* directory located in *listings* application and add the following code to it:

/finesauces/listings/templates/product/detail.html
```
{% extends 'base.html' %}

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

{% block content %}
<div class="card shadow-custom border-0 col-lg-8 mx-auto">
  <div class="row justify-content-center">
    <div class="col-lg-6">
      <img src="{{ product.image.url }}" alt=""
        class="img-fluid rounded-0">
    </div>
    <div class="col-lg-6">
      <div class="top">
        <h2 class="mt-4 font-weight-bold text-grey">
          {{ product.name }}
        </h2>
         <div>
          <span class="font-weight-bold text-grey">
            Hotness:
          </span> {{ product.shu }} SHU
        </div>
        <div>
          <span class="font-weight-bold text-grey">
            Volume:
          </span> 100ml
        </div>
      </div>
      <div class="mt-3">
        {{ product.description }}
      </div>
      {% if product.available %}
        <div class="alert alert-success my-3 text-center">
          Available
        </div>
      {% else %}
        <div class="alert alert-danger my-3 text-center">
          Currently unavailable
        </div>
      {% endif %}
      <hr>
      <div class="review">
        <span class="font-weight-bold text-grey">
          Average rating: 4.5/5.0
        </span>
        <div class="mb-3 font-weight-bold">
```

```
          <a href="" class="text-warning">
            Write review
          </a>
        </div>
      </div>
      <div class="mb-4">
        <form>
          <span class="input-group add-to-cart shadow-custom">
            <div class="input-group-prepend">
              <span class="input-group-text
                font-weight-bold px-3 btn btn-block price">
                ${{ product.price }}
              </span>
             </div>
            <span id="minus" class="input-group-text button_minus px-4">
              -
            </span>
            <input type="number" value="1" class="form-control
              text-center px-3">
            <span id="plus" class="input-group-text button_plus px-4">
              +
            </span>
            <div class="input-group-append">
              <button class="btn btn-danger px-4 reduce_padding"
                type="submit" data-toggle="tooltip"
                data-placement="top" title="Add to cart">
                <i class="fas fa-shopping-cart"></i>
              </button>
            </div>
          </span>
        </form>
      </div>
    </div>
  </div>
</div>
{% endblock content %}
```

When you hover over product card on the landing page and click on the magnifying glass icon, you will be redirected to the product detail page:



Our users can read a little more about our products here. However, they can't leave their reviews at this stage, so let's fix that now.

3.5.6 Adding reviews

When a user clicks on *Write review* link, we could redirect them to a new page with the Review form. However, to have it little more interactive and engaging, it might be a better idea to utilize modal window in this case.

Let's open our *detail.html* template and add the following code right between * * and *{% endblock content %}* at the end of the file.

/finesauces/listings/templates/product/detail.html

```
#...
<!-- MODAL -->
<div class="modal" id="myModal">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title">Write your review</h5>
        <button class="close" data-dismiss="modal">×</button>
```

```
      </div>
      <form>
        <div class="modal-body">
           <div class="bg-grey">
            <div class="text-center pt-2">
              Rating:
              <div id="full-stars" class="d-inline text-center">
                <div class="rating-group">
                  <label class="rating__label" for="id_rating_0">
                    <i class="rating__icon rating__icon--star fa fa-star">
                    </i>
                  </label>
                  <input class="rating__input" name="rating"
                    id="id_rating_0" value="1" type="radio" required>

                  <label class="rating__label" for="id_rating_1">
                    <i class="rating__icon rating__icon--star fa fa-star">
                    </i>
                  </label>
                  <input class="rating__input" name="rating"
                    id="id_rating_1" value="2" type="radio" required>

                  <label class="rating__label" for="id_rating_2">
                    <i class="rating__icon rating__icon--star fa fa-star">
                    </i>
                  </label>
                  <input class="rating__input" name="rating"
                    id="id_rating_2" value="3" type="radio" required>

                  <label class="rating__label" for="id_rating_3">
                    <i class="rating__icon rating__icon--star fa fa-star">
                    </i>
                  </label>
                  <input class="rating__input" name="rating"
                    id="id_rating_3" value="4" type="radio" required>

                  <label class="rating__label" for="id_rating_4">
                    <i class="rating__icon rating__icon--star fa fa-star">
                    </i>
                  </label>
                  <input class="rating__input" name="rating"
                    id="id_rating_4" value="5" type="radio" checked required>
                </div>
              </div>
            </div>
             <hr class="m-0">
            <div class="input-field text-center p-2">
              <textarea cols="40" rows="6"
                class="form-control shadow px-2">
              </textarea>
            </div>
          </div>
        </div>
        <div class="modal-footer">
          <button class="btn btn-danger" type="submit">Confirm</button>
        </div>
      </form>
    </div>
  </div>
</div>
#...
```

Then locate this line:

/finesauces/listings/templates/product/detail.html

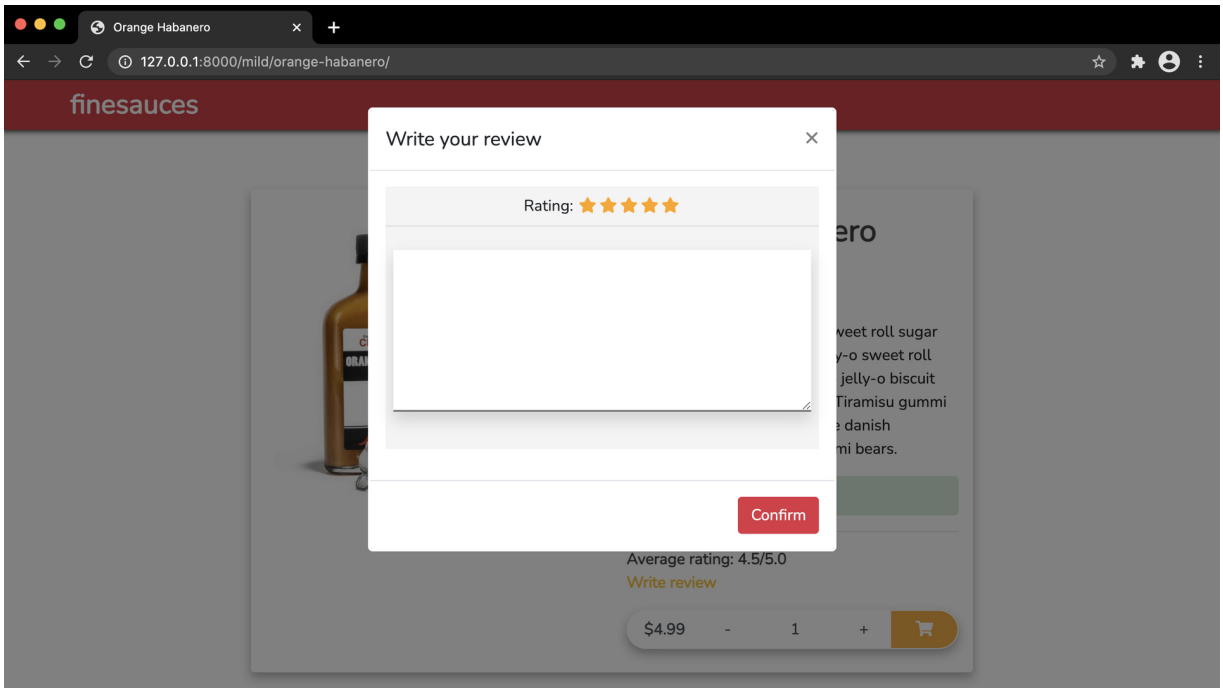
```
<a href="" class="text-warning">
  Write review
</a>>
```

And update it the following way so we are able to display our modal form:

/finesauces/listings/templates/product/detail.html

```
<a href="" class="text-warning" data-toggle="modal" data-target="#myModal" >
  Write review
</a>
```

We are now able to toggle modal window containing our simple review form, which users will be able to use to submit their feedback:



To store our product reviews in the database, we will need to define *Review* model. Open *models.py* file inside *listings* application and add the following code for our new model:

```
/finesauces/listings/models.py
from django.core.validators import MinValueValidator, MaxValueValidator
# ...
class Review(models.Model):
    product = models.ForeignKey(
        Product,
        related_name='reviews',
        on_delete=models.CASCADE
    )
    author = models.CharField(max_length=50)

    rating = models.IntegerField(
        validators=[MinValueValidator(1), MaxValueValidator(5)]
    )

    text = models.TextField(blank=True)
    created = models.DateTimeField(auto_now_add=True)

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

We start by importing *MinValueValidator* and *MaxValueValidator* validators. They will be instrumental in validating input from the review form that is being submitted (for example, if someone tried to submit an invalid score value of 20 or -100 for that matter). We will take advantage of them soon when we create our first form. As for other model attributes:

- ***product*** - we set up a many-to-one relationship (similar to *Category - Product*) between *Review* and *Product* models. One product can have multiple reviews, but each review belongs to a specific product.
- ***author*** - we want to display the first name of the review author.
- ***rating*** - will be integer field between 1 and 5.
- ***text*** - we are using *TextField* here instead of *Charfield* since we do not want to limit the text length
- ***created*** - We also want to track when the review was created. We specify it to be a date by using a *DateTimeField* attribute. We set the creation date by using the *auto_now_add* argument [26] . It's very useful for creating timestamps. If we wanted to keep track of the "last-modified" timestamp, we could use *the auto_now argument instead, as it is* automatically updated whenever the record is updated.

In *Meta* class, we specify that reviews should be sorted by descending order, from latest to earliest.

Let's create a new migrations file for our new model. Close the development server if it's running and run the *makemigrations* command:

```
terminal
(env) ~/finesauces$ python manage.py makemigrations
```

You should see output similar to this:

```
terminal
Migrations for 'listings':
   listings/migrations/0002_auto_20210102_2244.py
    - Change Meta options on category
    - Create model Review
```

Now run *migrate* command to create a table for our new model:

```
terminal
(env) ~/finesauces$ python manage.py migrate
```

After migrating our model, you will see the following message:

```
terminal
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, listings, sessions
Running migrations:
  Applying listings.0002_auto_20210102_2244... OK
```

To accept user input, we will need to build a review form. Luckily, we don't have to create everything from scratch, as Django offers very helpful functionality to assist us. Inside your *listings* application, create *forms.py* file:

```
/finesauces/listings/forms.py
from django import forms
from .models import Review

REVIEW_CHOICES = [('1', '1'), ('2', '2'), ('3', '3'), ('4', '4'), ('5', '5')]

class ReviewForm(forms.ModelForm):

    class Meta:
        model = Review
        fields = ['text', 'rating']

        widgets = {
            'text': forms.Textarea(
                attrs={
                    'class': 'form-control shadow px-2',
                    'rows': 6
                 }
            ),
            'rating': forms.RadioSelect(
                choices=REVIEW_CHOICES
             )
        }
```

REVIEW_CHOICES list represents pairs for each of the options we want to include in the *RadioSelect* input. They reflect our rating system from 1 to 5. Value in the first position of the pair defines a value that will be passed to the view to be processed. Value in the second position is used just for display in the HTML template. We will get back to it later in this section.

To create a form from a model, we just need to indicate which model we want the form to be based on. Django inspects the model and builds the form dynamically for us. By default, Django builds a form field for each field contained in the model. However, we can explicitly tell the framework which fields we want to include in our form using a fields list or define which fields we want to exclude using an exclude list of fields (*exclude = ['text', 'rating']*). We can also provide custom styling classes and other attributes here inside the *widgets* dictionary. We just need to specify the elements and attributes we would like to include. In this form, we specify custom styling classes *form-control, shadow* and *px-2* , which come from *Bootstrap* framework. Our *textarea* input field should also have a height of 6 rows and *rating* attribute should be generated as a group of radio buttons with values set in *REVIEW_CHOICES*, from *1* to *5*.

Each field type has a default widget that determines how the field is rendered in HTML. In *widgets* settings, we specify to render *text* attribute as *<textarea>* HTML element instead of default *<input>* element. These widget attributes will also have implications during form validations, where Django checks whether valid data was submitted. For example, if we specified a certain field to be of email type, Django will build <input> field with type email and expect email value to be submitted.

Now that our review form is created, we will need to process it somehow. We will utilize our *product_detail* view, which is already in place. Open the *views.py* file inside your *listings* application and add additional imports on top:

```
/finesauces/listings/views.py
from django.shortcuts import render, get_object_or_404, redirect
from .models import Category, Product, Review
from .forms import ReviewForm
```

make the *product_detail* view look like this:

```
/finesauces/listings/views.py
def product_detail(request, category_slug, product_slug):
    category = get_object_or_404(Category, slug=category_slug)
    product = get_object_or_404(
        Product,
        category_id = category.id,
        slug=product_slug
    )

    if request.method == 'POST':
        review_form = ReviewForm(request.POST)

        if review_form.is_valid():
            cf = review_form.cleaned_data

            author_name = "Anonymous"
            Review.objects.create(
```

```
                product = product,
                author = author_name,
                rating = cf['rating'],
                text = cf['text']
            )

        return redirect(
            'listings:product_detail',
            category_slug=category_slug, product_slug=product_slug)

    else:
        review_form = ReviewForm()

         return render(
            request,
            'product/detail.html',
            {
                'product': product,
                'review_form': review_form
            }
        )
```

We start by importing *redirect* function. Instead of simply using *render* function to show an updated detail page, we want to reroute a user back to the product detail page to avoid potentially performing the review submission twice if the user refreshed the page right after publishing the review. Then we import our *Review* model and *ReviewForm* that we just built.

In this new *product_detail* view, we differentiate between *POST* and *GET* requests. *POST* request will occur when a user submits a new review. If a user just visits the page, *GET* request will take place. In this case, we render the page within the product *detail.html* template, including *ReviewForm()*.

When *POST* request occurs, we perform the following actions:

- Instantiate *ReviewForm* with submitted data.

- Check whether the form is valid. That's where our *MinValueValidator* and *MaxValueValidator* do their part. This method validates data submitted in the form and returns True if all fields contain valid data. If any of the fields contain invalid data, we redirect the user back to the page without creating a Review object.

- If the form is valid, we retrieve validated data by accessing the *.cleaned_data* attribute of the form. This attribute is a dictionary of form fields and their values. This means we can access our submitted data using basic dictionary notations.

- Then we create a new *Review* object with data provided and save it to the database. Note that the *author* attribute defaults to "*Anonymous*" for now. Once we implement user accounts, we will modify this accordingly.

- Finally, we redirect the user back to our product detail page. Our *product_detail* view then handles the HTTP request with *GET* request.method.

Let's get back to *detail.html* and include our *ReviewForm* in a modal window. Find the opening form tag in our modal window section:

/finesauces/listings/templates/product/detail.html

```
#...
<form>
  <div class="modal-body">
    <div class="bg-grey">
    #...
```

add the following code to it:

/finesauces/listings/templates/product/detail.html

```
#...
<form method="post" >
  {% csrf_token %}
  <div class="modal-body">
    <div class="bg-grey">
    #...
```

We indicate that this form is to be submitted by the *POST* method. We also include template tag *{% csrf_token %}*, which introduces a hidden field containing an autogenerated token to avoid **cross-site request forgery (CSRF)** attacks. As per Django Cross-Site Request Forgery protection documentation [27] : "*This type of attack occurs when a malicious website contains a link, a form button or some JavaScript that is intended to perform some action on your website.*"

There are a couple of basic methods for rendering Django forms inside HTML templates [28] :

- ***as_p*** - this will tell Django to render form fields wrapped inside *<p>* tags
- ***as_ul*** - will render fields wrapped in *<li>* tags inside unordered <ul> list
- ***as_table*** - will render them as table cells wrapped in *<tr>* tags inside *<table>*

For the first method, syntax, including *review_form*, would be *{{ review_form.as_p }}*.

We can also iterate through our form fields:

```
{% for field in review_form %}
  <div>
    {{ field.errors }}
    {{ field.label_tag }} {{ field }}
```

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

And we can access specific form fields manually if we like to. Each field is available as an attribute of the form using *{{ form.name_of_field }}*, and in a Django template, will be rendered appropriately. This approach will also allow us to render fields in a different order, and we will use it to render our *review_form* , since this form has custom styling that would be hard to render using one of the previous techniques.

Open the *detail.html* template and find the following line within the *Modal* window section:

/finesauces/listings/templates/product/detail.html

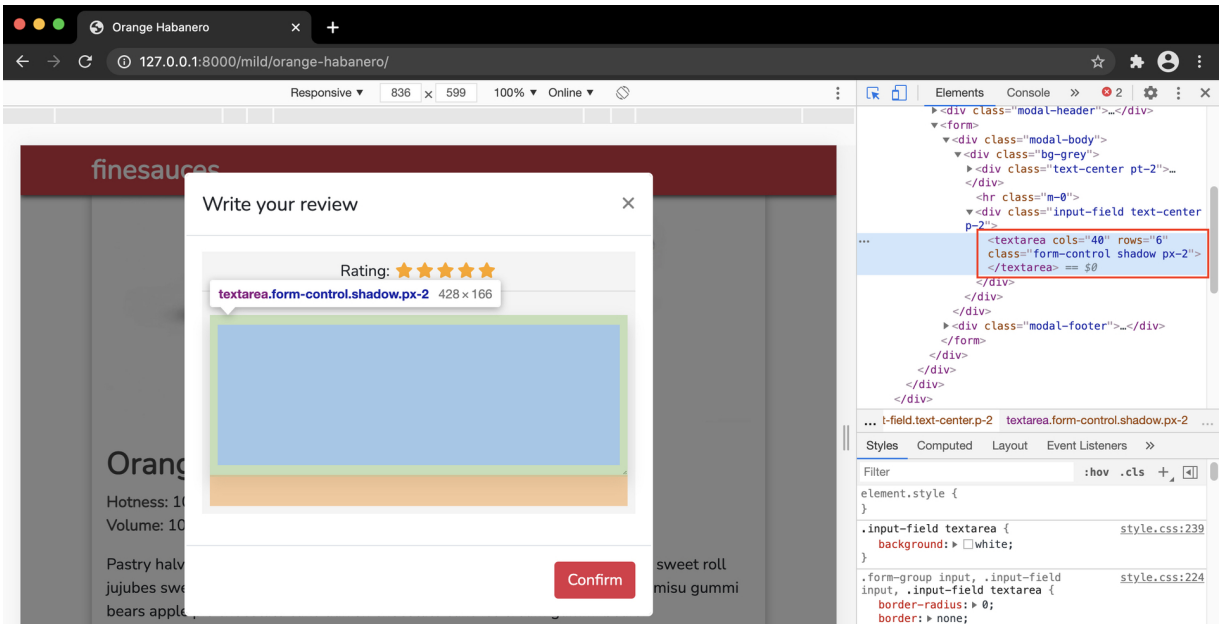
```
#...
<textarea cols="40" rows="6"
  class="form-control shadow px-2">
</textarea>
#...
```

replace it by:

/finesauces/listings/templates/product/detail.html

```
#...
{{ review_form.text }}
#...
```

Let's go ahead and execute *python manage.py runserver* command to start the development server. Open http://127.0.0.1:8000/ in your browser and go to detail page of any of your products. Click on the Write review to open up the modal form and then access *Developer tools* by either pressing *F12* button or right-clicking the page and selecting the *Inspect* option. Select the *Inspect* tool by clicking on the cursor on the top of the *Developer tools* section on the left side of the *Elements*, *Console*, and *Sources* fields. Then focus on our *Textarea* field. You will be able to see detailed information about how Django generated this field. You should see output similar to this:



Class and *rows* attributes come from our *ReviewForm widgets* located inside *forms.py* file. As for retrieving and instantiating form fields, Django is concerned only about this element's name and id attributes. The implication of this fact would be that if you want to build your own forms from scratch and not rely on Django but still take advantage of *form.is_valid()* validation, you can do so if you provide valid names and ids for the elements. Element *name* parameter would be the attribute *name* specified in the *models.py* file for the given model you want to handle, *id* would then be composed of prefix *id_* and *name* of the attribute, so for *text* attribute, it would be *id_text* .

As with everything, there might be some exceptions or deviations from this rule. Let's have a look at our stars rating. It is still implemented as one element, *Radioselect* . However, each star represents another element in a sense.

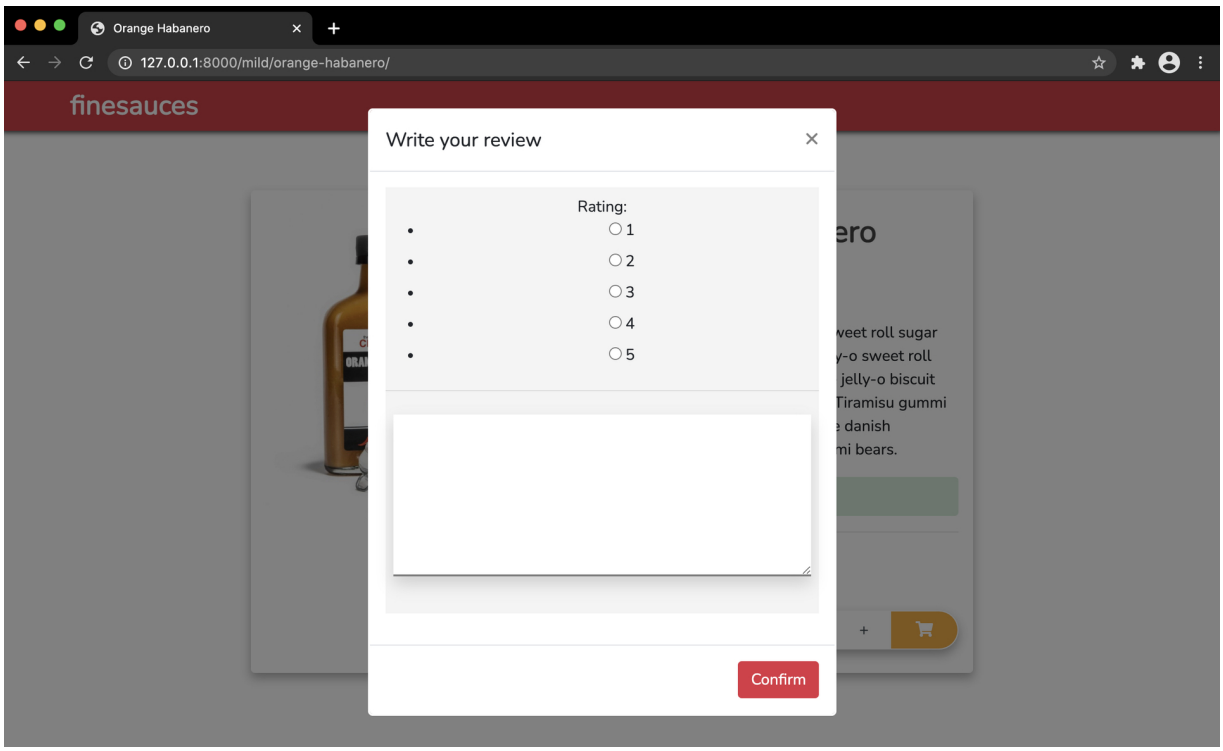
To explore this in detail, go ahead and open your *detail.html* file. Then modify the modal form to look like this:

/finesauces/listings/templates/product/detail.html

```
#...
<form method="post">
  {% csrf_token %}
  <div class="modal-body">
    <div class="bg-grey">
      <div class="text-center pt-2">
        Rating:
        {{ review_form.rating }}
      </div>
      <hr class="m-0">
      <div class="input-field text-center p-2">
        {{ review_form.text }}
      </div>
    </div>
  </div>
```

```
  <div class="modal-footer">
    <button class="btn btn-danger" type="submit">Confirm</button>
  </div>
</form>
#...
```

We basically swapped out the whole stars section with one line *{{ review_form.rating }}*. Go ahead and click on Write review to open the review form inside the modal window. You will be welcomed by very basic, not very user-friendly *RadioSelect* form:



To access individual elements within our RadioSelect, we can utilize indexing like this:

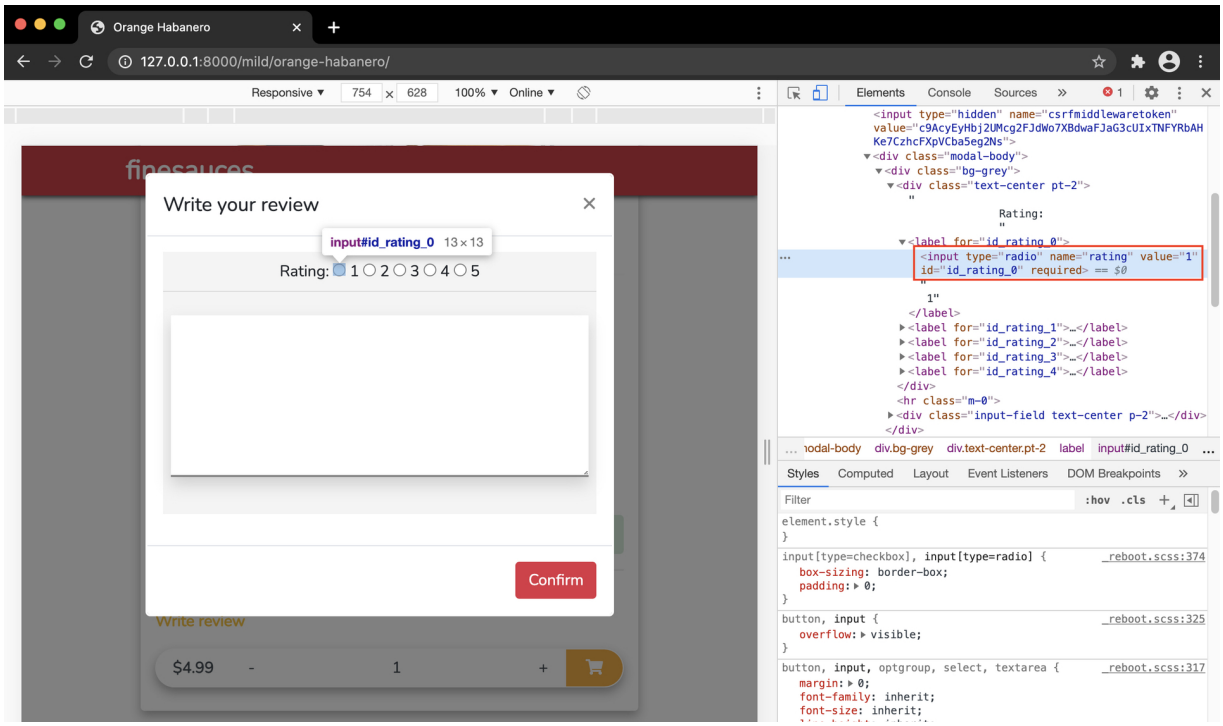
```
{{ review_form.rating.0 }}
```

to get hold of the first element. Let's update the rest of the elements as well. We will end up with the following solution:

/finesauces/listings/templates/product/detail.html

```
#...
<div class="text-center pt-2">
  Rating:
  {{ review_form.rating.0 }}
  {{ review_form.rating.1 }}
  {{ review_form.rating.2 }}
  {{ review_form.rating.3 }}
  {{ review_form.rating.4 }}
</div>
#...
```

Go ahead and inspect that review form as we did previously. Focus on the first element. You should see something like this:



Little better. Regardless of the look, this form is very vital for understanding how Django forms work.

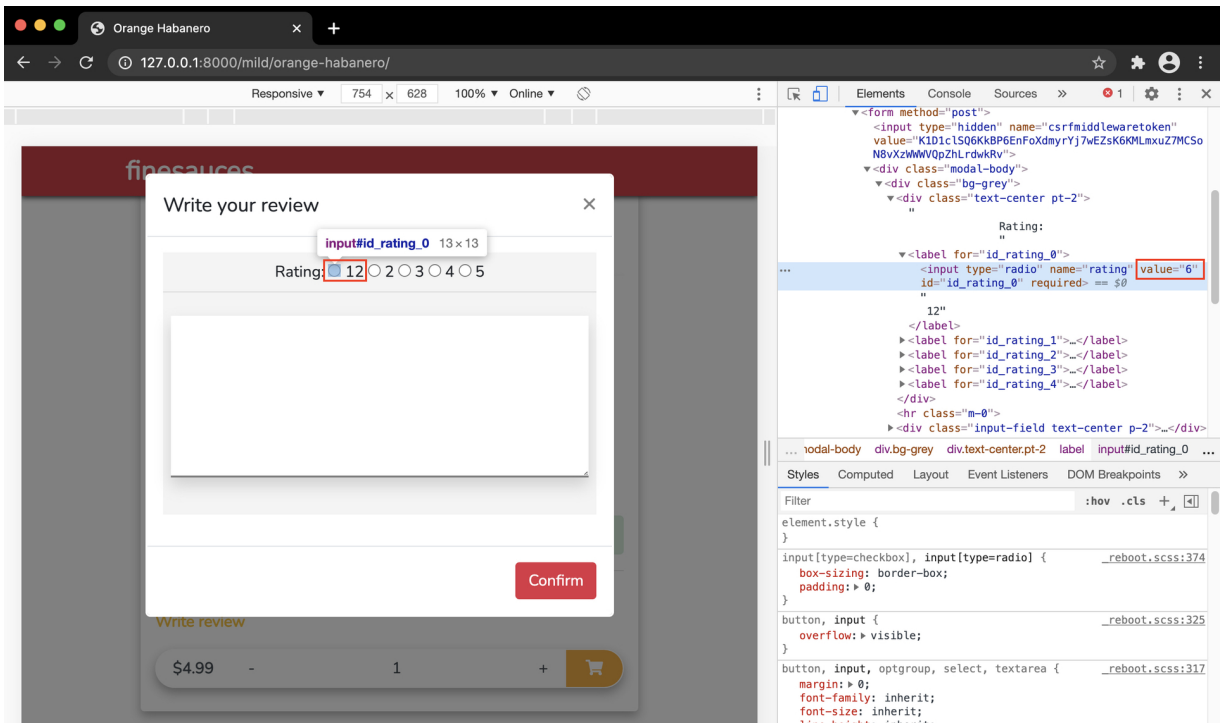
For the first radio select element, we see the following attributes:

```
<input type="radio" name="rating" value="1" id="id_rating_0" required="">
```

Like the *text* field, the *name* attribute follows the same pattern, the *name* of the model field. For the *id* , however, Django appends the index at the end after the model field name. For the first element, it is 0, for the second 1, for the third 2, and so on. The *value* "1" for the first element comes from the *forms.py REVIEW_CHOICES* list first tuple. Remember, that value on the first position in that tuple specifies the *form element's value attribute* . The value on the second position then specifies the value displayed on the HTML page. If you changed the *REVIEW_CHOICES* list to:

```
REVIEW_CHOICES = [('6', '12'), ('2', '2'), ('3', '3'), ('4', '4'), ('5', '5')]
```

values for the first radio select element would change accordingly:



Knowing how Django generates and handles forms and what it expects when validating them enables us to start building our custom form templates if we need/want to.

Return to our detail.html template and replace:

```
/finesauces/listings/templates/product/detail.html
#...
{{ review_form.rating.0 }}
{{ review_form.rating.1 }}
{{ review_form.rating.2 }}
```

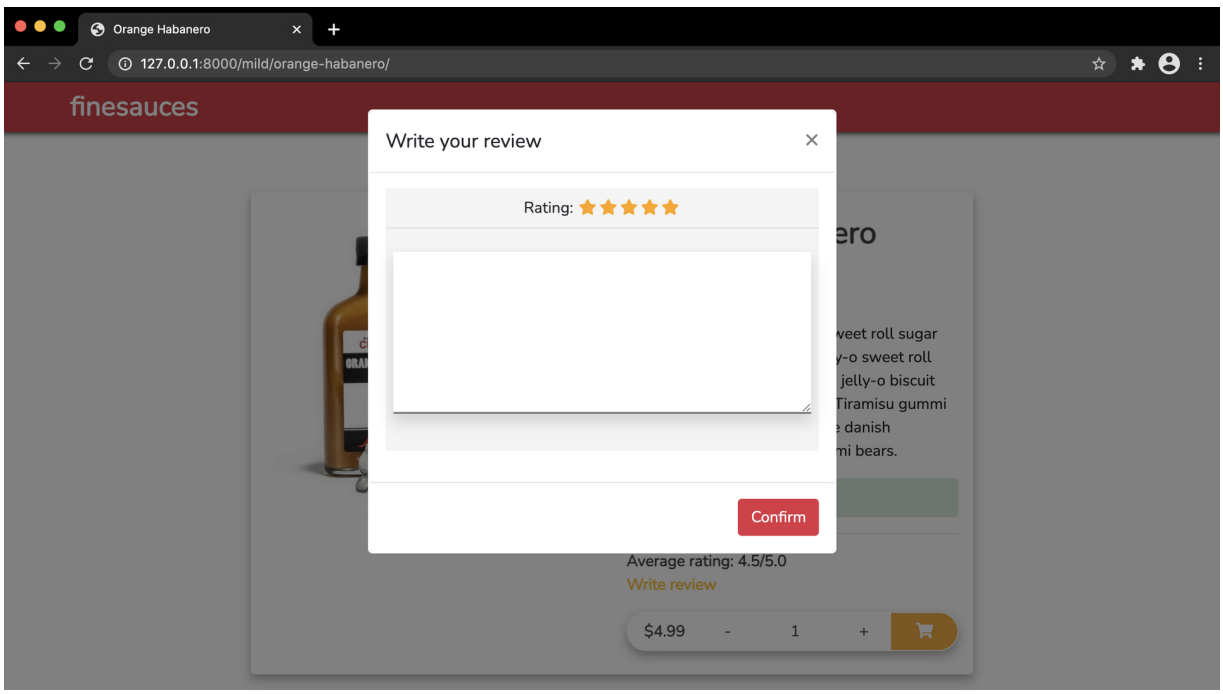
```
{{ review_form.rating.3 }}
{{ review_form.rating.4 }}
#...
```

with

/finesauces/listings/templates/product/detail.html

```
#...
<div id="full-stars" class="d-inline text-center">
  <div class="rating-group">
     <label class="rating__label" for="id_rating_0">
       <i class="rating__icon rating__icon--star fa fa-star"></i>
    </label>
    <input class="rating__input" name="rating"
      id="id_rating_0" value="1" type="radio" required>

    <label class="rating__label" for="id_rating_1">
      <i class="rating__icon rating__icon--star fa fa-star"></i>
    </label>
    <input class="rating__input" name="rating"
      id="id_rating_1" value="2" type="radio" required>

    <label class="rating__label" for="id_rating_2">
      <i class="rating__icon rating__icon--star fa fa-star"></i>
    </label>
    <input class="rating__input" name="rating"
      id="id_rating_2" value="3" type="radio" required>

    <label class="rating__label" for="id_rating_3">
      <i class="rating__icon rating__icon--star fa fa-star"></i>
    </label>
    <input class="rating__input" name="rating"
      id="id_rating_3" value="4" type="radio" required>

    <label class="rating__label" for="id_rating_4">
      <i class="rating__icon rating__icon--star fa fa-star"></i>
    </label>
    <input class="rating__input" name="rating"
      id="id_rating_4" value="5" type="radio" checked required>
  </div>
</div>
#...
```

Now we have proper, good looking stars rating available:



With manually creating our form template, we are not taking advantage of Django autogenerating the form for us. This means we can actually remove *REVIEW_CHOICES* from *forms.py* file. Let's also adjust the *rating* widget to look like this:

/finesauces/listings/forms.py

```
#...
'rating': forms.RadioSelect
#...
```

Before submitting our first review, we have to let the Django admin site know about our *Review* model. Open *admin.py* file inside your *listings* application and add the following code to it:

```
/finesauces/listings/admin.py
from django.contrib import admin
from .models import Category, Product, Review

class OrderReviewInline(admin.TabularInline):
    model = Review
    #...
```

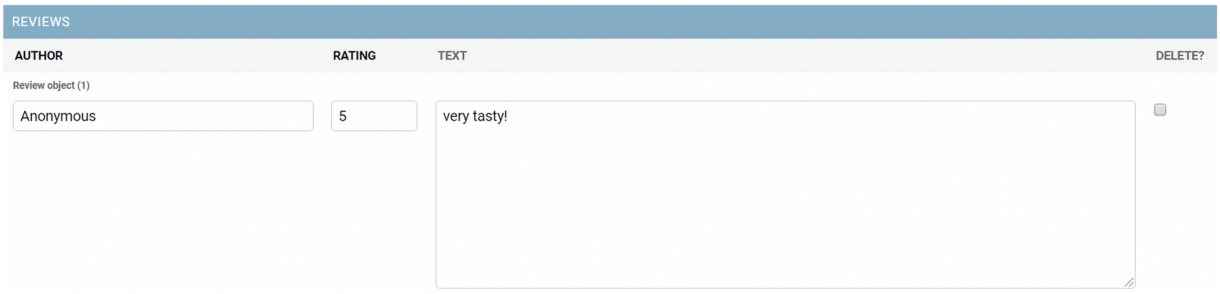
Also add *inlines = [OrderReviewInline]* to the end of the *ProductAdmin* class, after prepopulated_fields:

```
/finesauces/listings/admin.py
#...
@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    #...
    prepopulated_fields = {'slug': ('name',)}

    inlines = [OrderReviewInline]
```

We are using a *TabularInline* subclass of *InlineModelAdmin* for the *Review* model to include it as an inline in the *ProductAdmin* class. An inline allows us to include a model on the same edit page as its related model.

Let's go ahead and submit our first review. Click on the *Write review* , fill in the form, and click on *Confirm* . You will then be redirected back to the detail page, but no reviews are visible yet. To confirm that our review actually went through, navigate to the admin site http://127.0.0.1:8000/admin/ . Go to the products section and select the product for which you wrote a review.

Below product information, you should see the *Review* section with your review:



To display product reviews on the product detail page, we have to make some updates to the *detail.html* template. Open the file and append the following code between * * and *<!-- MODAL -->* , so replace:

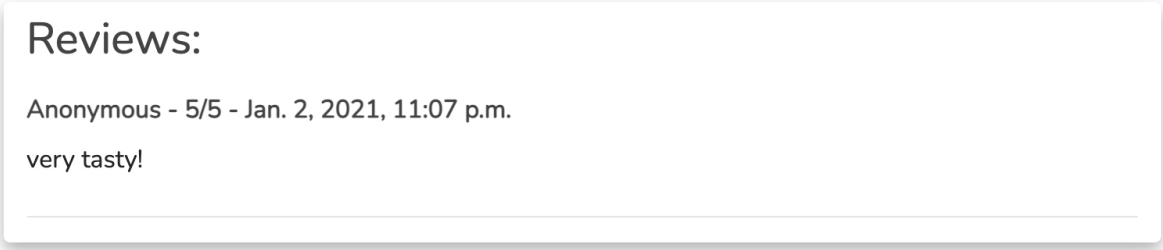
```
/finesauces/listings/templates/product/detail.html
#...
</div>
<!-- MODAL -->
<div class="modal" id="myModal">
#...
```

with:

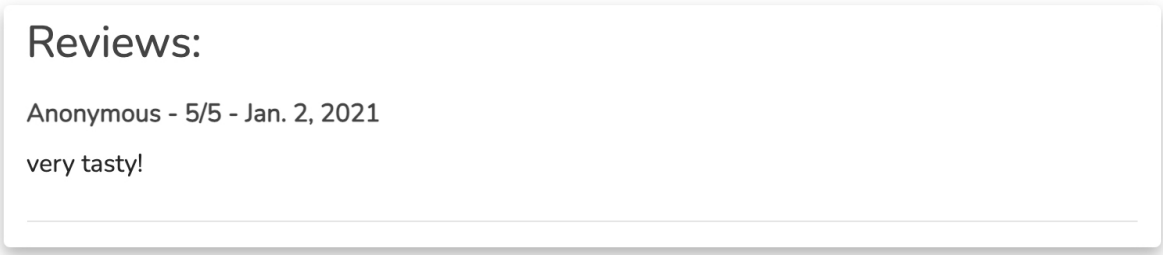
```
/finesauces/listings/templates/product/detail.html
#...
</div>
<hr class="col-7 mx-auto">
<div class="card shadow-custom border-0 col-lg-8 mx-auto mb-3">
  <h3 class="py-2 text-grey">Reviews:</h3>
  {% for review in product.reviews.all %}
    <span class="font-weight-bold py-2 text-grey">
      {{ review.author }} - {{ review.rating }}/5 - {{ review.created }}
    </span>
    <span class="text-justify pb-2">
      {{ review.text }}
    </span>
    <hr>
  {% empty %}
    <span class="pb-2">Currently no reviews</span>
  {% endfor %}
</div>
<!-- MODAL -->
<div class="modal" id="myModal">
#...
```

We use *{% for %}* template tag to iterate through the product reviews. In case there are no reviews, we display a default message after *{% empty %}* tag.

This is the first comment for our product:



We can see our default user name, rating, date of writing this review, and a comment. If you would like to show date without a specific time, you can utilize a template filter *date* by updating *{{ review.created }}* to *{{ review.created |**date** }}:*



The last thing remaining to implement in this chapter would be the average rating calculation for our products. We will achieve this by creating a simple function inside our *Product* model. Open *models.py* file inside *listings* application and add the following code to the *Product* model:

/finesauces/listings/models.py

```
#...
def get_average_review_score(self):
    average_score = 0.0
    if self.reviews.count() > 0:
        total_score = sum([review.rating for review in self.reviews.all()])
        average_score = total_score / self.reviews.count()
    return round(average_score, 1)
    #...
```

We use *count()* to get the total number of objects, in our case reviews. If there are any reviews available, we iterate through the *QuerySet* to get the total sum of review ratings, which we then divide by the number of reviews to get the average rating for the product. We then return this score as a number with 1 decimal place.

Let's update *detail.html* template to show the product average score. Replace the hardcoded 4.5 rating:

/finesauces/listings/templates/product/detail.html

```
<span class="font-weight-bold text-grey">
  Average rating: 4.5/5.0
</span>
```

with variable:

/finesauces/listings/templates/product/detail.html

```
<span class="font-weight-bold text-grey">
  Average rating: {{ product.get_average_review_score }} /5.0
</span>
```

Refresh your product detail page for the update to take effect. Feel free to add a couple of reviews to see how the score changes.

3.6 Local and remote repository

It's a good idea to back up your progress after every major update to the code. Before pushing a new version of our project to our remote repository at *Bitbucket* , let's commit those updates to the local repository first. Run the *git add -A* command to include all of the files and their updates present in the project directory in the new commit:

terminal

```
(env) ~/finesauces$ git add -A
```

Now commit those files by also providing a short message describing what changed:

terminal

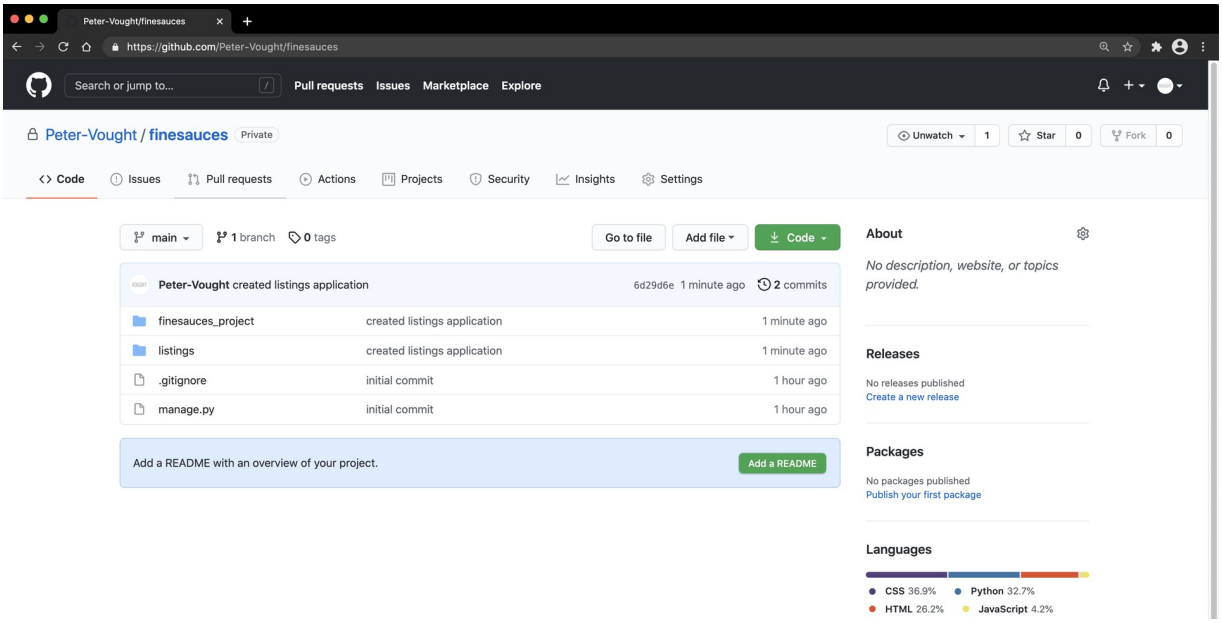
```
(env) ~/finesauces$ git commit -m "created listings application"
```

To push the latest project version to our remote repository, run:

terminal

```
(env) ~/finesauces$ git push -u origin master
```

Visit *Github* repository to see the latest project version successfully uploaded:



4 Shopping cart

In the previous section, we finished building our *listings* application. Users can now browse through our products, view product detail pages, and leave a review. The next step will be to enable them to add products to the shopping cart.

The shopping cart will be persisted in the session, so the cart content is preserved throughout the user's visit. We will implement this by utilizing Django's session framework. The cart will be kept in the session until the session finishes, or the cart is checked out.

4.1 Sessions

Django provides full support for anonymous sessions. The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It stores data on the server-side and abstracts the sending and receiving of cookies. Cookies contain a session ID – not the data itself [29] .

Sessions are enabled by default when we start our project using *startproject* command. We can find the sessions enabled and in the *settings.py* file inside *MIDDLEWARE* list as *'django.contrib.sessions.middleware.SessionMiddleware'*. *and* in *INSTALLED_APPS* list as *'django.contrib.sessions'* . When *SessionMiddleware* is activated, each *HttpRequest* object that our views receive as a first argument will have a *session* attribute, which is a dictionary-like object. We can use dictionary notations to store and retrieve its data.

We can set our variables in session the following way:

```
request.session['cart_id'] = {}
```

which will create our empty cart in the session. We can retrieve this cart by using:

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

and delete a cart:

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

This is enough to get us started. Sessions are, however, much broader topic than presented here. For a complete list of available settings and options, please refer to the Django documentation [30] .

4.2 Storing shopping cart in session

Our cart will need to be able to store the following information about individual products:

- ID of the product
- Quantity of the product
- Unit price of the product

In theory, we wouldn't need to store the unit price for the product. We could easily get it from our database. However, by storing the unit price in the session, we make sure our customer buys the product for the intended price, which was current when added to the cart, regardless of later fluctuations.

Before implementing our shopping cart, let's edit the *settings.py* file and add the *CART_ID* to the bottom of the file, in front of the *local_settings* import:

/finesauces/finesauces_project/settings.py
```
#...
CART_ID = 'cart'
#...
```

We will use this key to store and retrieve our shopping cart from the session. Now we need to create our dedicated cart application. Open your terminal, and run the following command:

terminal
```
(env) ~/finesauces$ python manage.py startapp cart
```

Django needs to know about our new application. Add it to the *INSTALLED_APPS* list located in *settings.py* file:

/finesauces/finesauces_project/settings.py
```
#...
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'listings.apps.ListingsConfig',
    'cart.apps.CartConfig',
]
#...
```

4.3 Shopping cart form and views

Before working on our cart views, let's go ahead and create our form so we are able to add items to the cart in the first place. Create *forms.py* file inside the *cart* application and add the following code to it:

/finesauces/cart/forms.py
```
from django import forms
from django.forms import NumberInput

class CartAddProductForm(forms.Form):
    quantity = forms.IntegerField(
        min_value=1,
        widget=NumberInput(attrs={
            'class': 'form-control text-center px-3',
            'value': 1
        }))
```

Our form is pretty straightforward. It contains only *quantity* field, which will be used to handle the quantity of products that are supposed to be added to the cart.

The first view we are going to create will be responsible for retrieving our cart from the session. This view will create an empty cart and store it in the session if no cart is found. Open the *views.py* file in the *cart* application and update it to look like this:

/finesauces/cart/views.py
```
from django.conf import settings

def get_cart(request):
    cart = request.session.get(settings.CART_ID)
    if not cart:
        cart = request.session[settings.CART_ID] = {}
    return cart
```

Next view will be used to add requested products to the cart. Add the following imports and the *cart_add* view to the *views.py* file:

/finesauces/cart/views.py
```
from django.shortcuts import render, redirect, get_object_or_404
from listings.models import Product
from .forms import CartAddProductForm
from decimal import Decimal

def cart_add(request, product_id):
    cart = get_cart(request)
    product = get_object_or_404(Product, id=product_id)
    product_id = str(product.id)
    form = CartAddProductForm(request.POST)
    if form.is_valid():
        cd = form.cleaned_data

        if product_id not in cart:
            cart[product_id] = {
                'quantity': 0,
                'price': str(product.price)
            }

         if request.POST.get('overwrite_qty'):
            cart[product_id]['quantity'] = cd['quantity']
        else:
            cart[product_id]['quantity'] += cd['quantity']

        request.session.modified = True

        return redirect('cart:cart_detail')
```

We start by retrieving our cart from the session. We then retrieve the requested product object and convert that product's id to string since Django uses JSON to serialize session data, and JSON only allows string key names. Then we instantiate and validate our *CartAddProductForm* form. If the given product id is not already in the cart, we create new product item with 0 initial quantity and current product price, which is also converted to the string to be serialized. In this view, we also look for '*overwrite_qty*' value, which serves us as an indicator of whether the current product quantity in the cart should be overwritten with a new quantity submitted in the form, or they should be added together. We then mark the session as modified [31] to be saved by Django.

Now that we are able to add items to the cart, we can work on displaying its contents. Add the following *cart_detail* view to the *views.py* file:

/finesauces/cart/views.py
```
def cart_detail(request):
    cart = get_cart(request)
    product_ids = cart.keys()
    products = Product.objects.filter(id__in=product_ids)
    temp_cart = cart.copy()

    for product in products:
```

```
        cart_item = temp_cart[str(product.id)]
        cart_item['product'] = product
        cart_item['total_price'] = (Decimal(cart_item['price'])
                                    * cart_item['quantity'])

     cart_total_price = sum(Decimal(item['price']) * item['quantity']
                        for item in temp_cart.values())

    return render(
        request,
        'detail.html',
        {
            'cart': temp_cart.values(),
            'cart_total_price': cart_total_price
        })
```

We start by retrieving our cart from the session. Then we get the list of our products in the cart by retrieving cart keys. Based on them, we query the product objects from the database. We then create a copy of our cart since we are going to temporarily add a couple of values there that we want to display in our templates, but we don't need to store in the session.

Then we iterate through our product objects present in our cart, and for each of these objects, we add the product instance and total price of the items to the temporary cart. We also calculate the total cost of the items in the shopping cart.

To remove items from the cart and clear the cart completely after the checkout, add the following two views to the file:

/finesauces/cart/views.py
```
#...
def cart_remove(request, product_id):
    cart = get_cart(request)
    product_id = str(product_id)
    if product_id in cart:
        del cart[product_id]

        request.session.modified = True

        return redirect('cart:cart_detail')

def cart_clear(request):
    del request.session[settings.CART_ID]
```

So far, we have created our *cart form* and views for managing cart quantity and displaying its contents. Let's proceed by creating URLs and mapping them to these views. Inside your *cart* application, create a new file named *urls.py* and add the following code to it:

/finesauces/cart/urls.py
```
from django.urls import path
from . import views

app_name = 'cart'

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

To let Django know about our new URL patterns, add them to the main *urls.py* file inside our *finesauces_project* directory:

/finesauces/finesauces_project/urls.py
```
#...
urlpatterns = [
    path('admin/', admin.site.urls),
    path('cart/', include('cart.urls', namespace='cart')),
    path('', include('listings.urls', namespace='listings')),
]
#...
```

To avoid any potential routing issues, remember to add your URL patterns above the `''` path leading to the landing page.

4.4 Displaying our cart

The only view that will display any template is *cart_detail* . Let's create the *templates* folder inside the *cart* application and *detail.html* file directly within it. Add the following code to the file:

/finesauces/cart/templates/detail.html
```
{% extends 'base.html' %}

{% block title %}Shopping cart{% endblock %}

{% block content %}
<div class="card shadow-custom border-0 col-lg-10 mx-auto mb-3">
  <h3 class="py-2 font-weight-bold text-grey text-center">
    Your order:
  </h3>
```

```
<div class="row justify-content-center py-2 text-center header-desc">
  <div class="col-lg-2 align-self-center">
    Product
  </div>
  <div class="col-lg-2 align-self-center">
    Description
  </div>
  <div class="col-lg-2 align-self-center">
    Availability
  </div>
  <div class="col-lg-1 align-self-center">
    Price
  </div>
  <div class="col-lg-2 align-self-center">
    Quantity
  </div>
  <div class="col-lg-3 align-self-center">
    Total price
  </div>
</div>
<hr class="mt-0">
{% for item in cart %}
  {% with product=item.product %}
    <div class="row justify-content-center cart">
      <div class="cart-item col-4 col-lg-2">
        <a href="{{ product.get_absolute_url }}">
          <img src="{{ product.image.url }}" alt=""
            class="img-fluid rounded-0">
        </a>
      </div>
    <div class="cart-item product-name col-12
      col-lg-2 align-self-center text-center">
      {{ product.name }}
    </div>
     <div class="cart-item col-12 col-lg-2 align-self-center">
    {% if product.available %}
      <div class="text-green mb-0 text-center">
        Available
      </div>
    {% else %}
      <div class="text-danger mb-0 text-center">
        Currently unavailable
      </div>
    {% endif %}
    </div>
    <div class="cart-item col-12 col-sm-3
      col-lg-1 align-self-center text-center">
      <span>${{ product.price }}</span>
    </div>
    <div class="cart-item col-6 col-sm-5 col-lg-2 align-self-center">
      <form>
        <span class="input-group shopping-cart">
          <input type="number" value="1"
            class="form-control text-center px-3">
          <input type="hidden" name="overwrite_qty" value="True">
          <div class="input-group-append">
            <button type="submit" class="btn text-white"
              data-toggle="tooltip" data-placement="top"
              title="Update">
              <i class="fas fa-edit"></i>
            </button>
          </div>
        </span>
      </form>
    </div>
    <div class="cart-item col-12 col-sm-3
      col-lg-2 align-self-center text-right">
      <span>${{ item.total_price }}</span>
    </div>
    <div class="cart-item col-12 col-lg-1 text-right
      align-self-center">
      <form>
        <button type="submit" class="btn" data-toggle="tooltip"
          data-placement="top" title="Remove">
          <i class="fas fa-trash-alt"></i>
        </button>
       </form>
    </div>
  </div>
  {% endwith %}
{% empty %}
<div class="align-self-center">
  <span class="text-center font-weight-bold text-muted lead">
    Your shopping cart is empty.
  </span>
</div>
{% endfor %}
<hr>
<div class="total-price-cart">
  <h4 class="row justify-content-center">
```

```
      <div class="col-5 col-lg-9 text-right text-grey">
        Total price:
      </div>
      <div class="col-5 col-lg-2 text-right text-danger">
        ${{ cart_total_price|floatformat:2 }}
      </div>
      <div class="col-2 col-lg-1">
      </div>
    </h4>
  </div>
</div>
<hr>
<div class="col-lg-10 mx-auto mb-5">
  <div class="row justify-content-end">
    <div class="col-lg-6 px-0">
      <div class="btn-group d-flex">
        <a href="{% url 'listings:product_list' %}"
          class="btn btn-warning shadow-custom col">
          Back to shop
        </a>
        <a href="" class="btn btn-danger shadow-custom col">
          Checkout
        </a>
      </div>
    </div>
  </div>
</div>
{% endblock content %}
```

This is the template that will be used to display our cart's content. It will allow users to change the quantity of selected products and remove them as well. We use template filter *floatformat* [32] to round the total cart cost to two decimal places.

4.5 Adding products to the cart

Users will be able to add products to the cart or modify the quantity from two locations. Product detail page and the cart itself. We shall start with updating the product detail page.

Update the *views.py* file located in the *listings* application to include the following import and modify *product_detail* view to handle our *CartAddProductForm* :

/finesauces/listings/views.py

```
from cart.forms import CartAddProductForm
#...
else:
    review_form = ReviewForm()
    cart_product_form = CartAddProductForm()

    return render(
        request,
        'product/detail.html',
        {
            'product': product,
            'review_form': review_form,
            'cart_product_form': cart_product_form
        }
    )
```

Open the *product/detail.html* template and locate the opening *<form>* tag:

/finesauces/listings/templates/product/detail.html

```
#...
<div class="mb-4">
  <form>
    <span class="input-group add-to-cart shadow-custom">
    #...
```

Modify the form action field to reference our *cart_add* URL. Don't forget to include *{% csrf_token %}* as well:

/finesauces/listings/templates/product/detail.html

```
#...
<div class="mb-4">
  <form action="{% url 'cart:cart_add' product.id %}" method="post" >
    {% csrf_token %}
    <span class="input-group add-to-cart shadow-custom">
    #...
```

We also need to replace the quantity field:

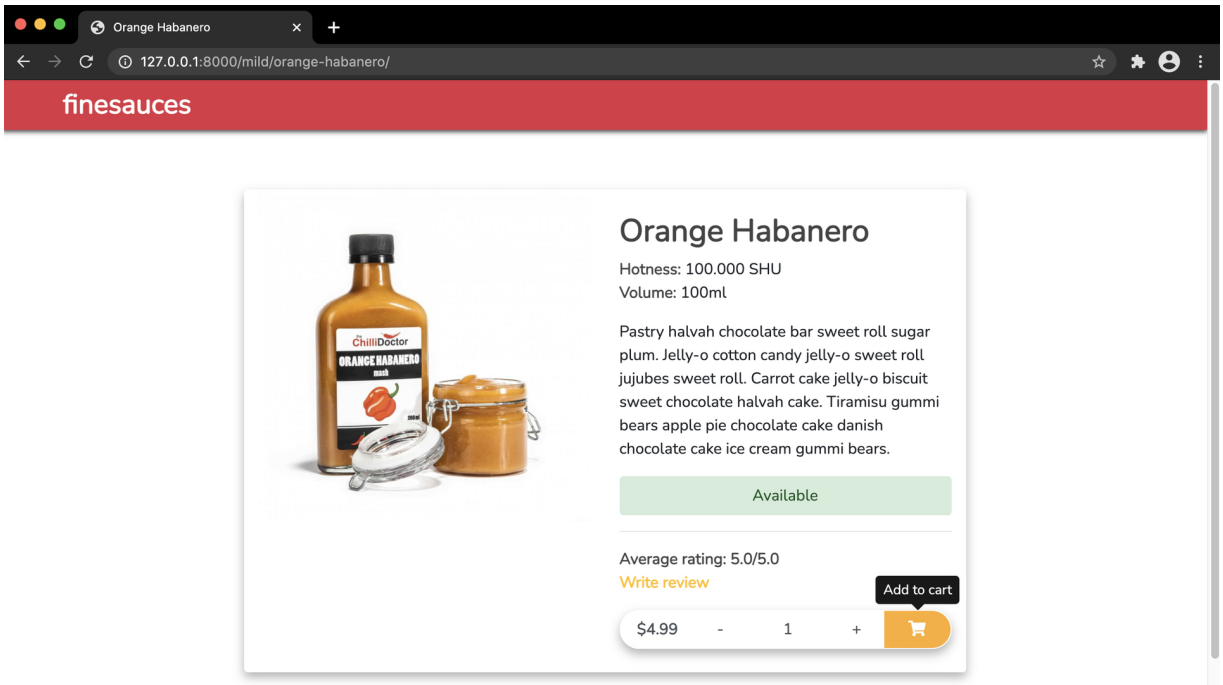
/finesauces/listings/templates/product/detail.html

```
#...
<input type="number" value="1" class="form-control
  text-center px-3">
#...
```

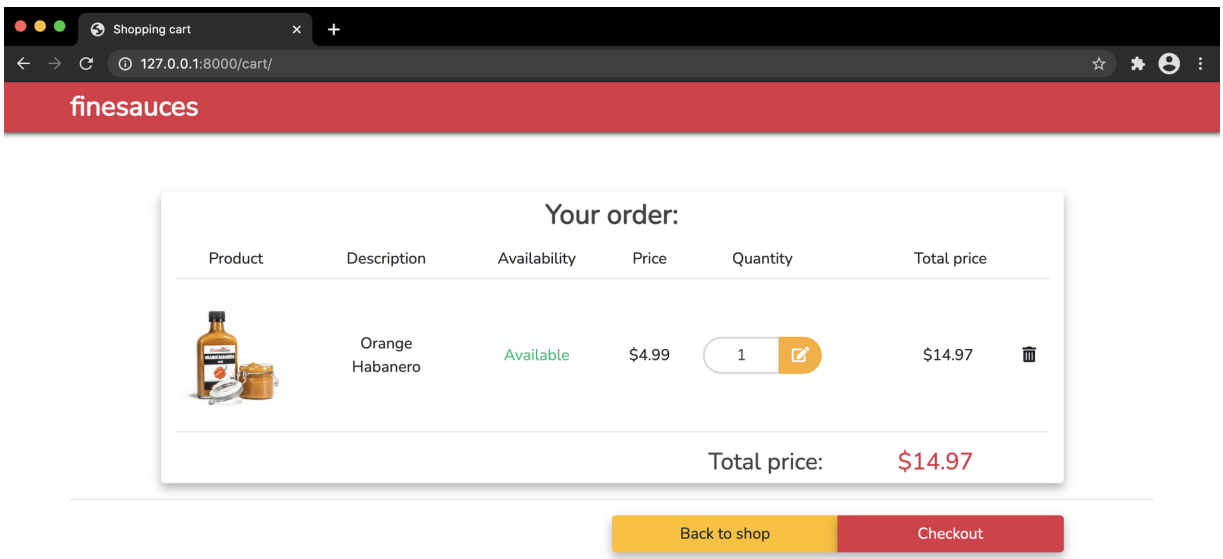
by:

```
/finesauces/listings/templates/product/detail.html
#...
{{ cart_product_form.quantity }}
#...
```

Make sure the development server is running with the *python manage.py runserver* command. Open http://127.0.0.1:8000 in your browser and access the detail page of one of your products. At first glance, nothing has changed. However, our form is now fully functional. Select your desired quantity and click on the *Add to cart* button:



The form is then submitted to the *cart_add* view. This view adds the requested product to the cart in the session, including its current price and the selected quantity. The user is then redirected to the cart detail page:



We can see the necessary product information on this page. Product image, product name, unit price and calculated total cost for 3 items. However, we still need to update forms in the cart *detail.html* template to manage product quantity and to be able to remove products from the cart as well.

4.6 Updating product quantities in cart

Let's start by modifying the form responsible for managing the product quantity in the cart. Open *views.py* file of the *cart* application and modify *cart_detail* view as follows:

```
/finesauces/cart/views.py
#...
def cart_detail(request):
    #...
    for product in products:
        cart_item = temp_cart[str(product.id)]
        cart_item['product'] = product
        cart_item['total_price'] = (Decimal(cart_item['price'])
                                    * cart_item['quantity'])
        cart_item['update_quantity_form'] = CartAddProductForm(initial={
            'quantity': cart_item['quantity']
        })
```

```
    cart_total_price = sum(Decimal(item['price']) * item['quantity']
                           for item in temp_cart.values())
    #...
```

We create an instance of *CartAddProductForm* for each item in the cart, so the user can update quantities. We initialize the form with the current item quantity present in the cart.

To activate this form, we need to update the cart *detail.html* template as well. Find the following lines:

/finesauces/cart/templates/detail.html

```
#...
<form>
  <span class="input-group shopping-cart">
    <input type="number" value="1"
      class="form-control text-center px-3">
      #...
```

And update it to reference our *cart_add* URL in the form action field, specify form method to *post* , include *{% csrf_token %}* and replace *<input>* field:

/finesauces/cart/templates/detail.html

```
#...
<form action="{% url 'cart:cart_add' product.id %}" method="post" >
  {% csrf_token %}
  <span class="input-group shopping-cart">
    {{ item.update_quantity_form.quantity }}
    #...
```

Now let's include our *remove* form as well, so the user can remove unwanted items from the cart. Find the following section:

/finesauces/cart/templates/detail.html

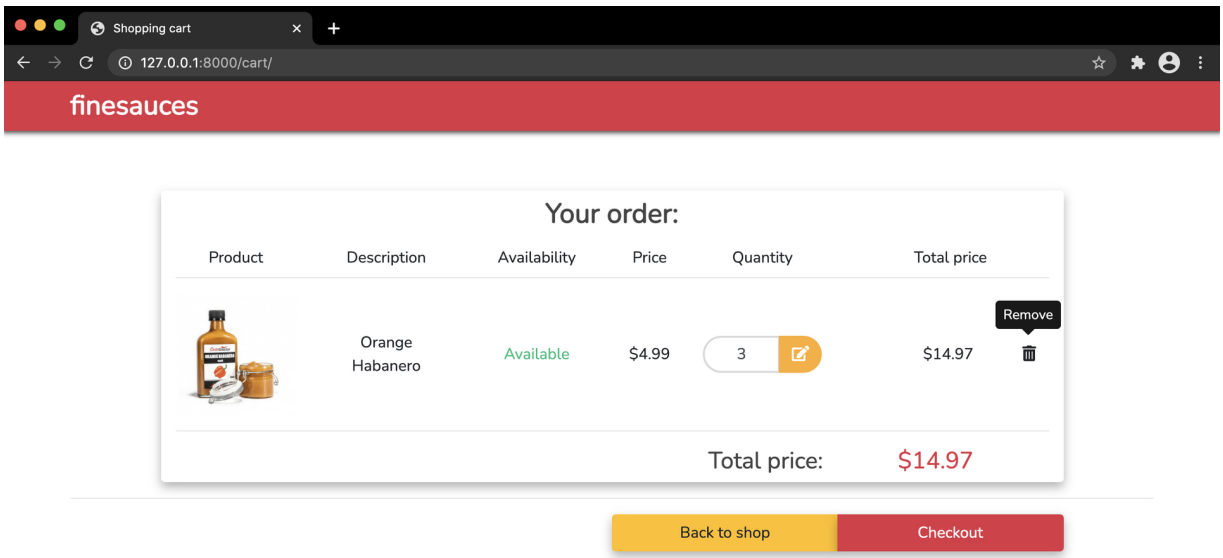
```
#...
<form>
  <button type="submit" class="btn" data-toggle="tooltip"
    data-placement="top" title="Remove">
    <i class="fas fa-trash-alt"></i>
    #...
```

And update it to:

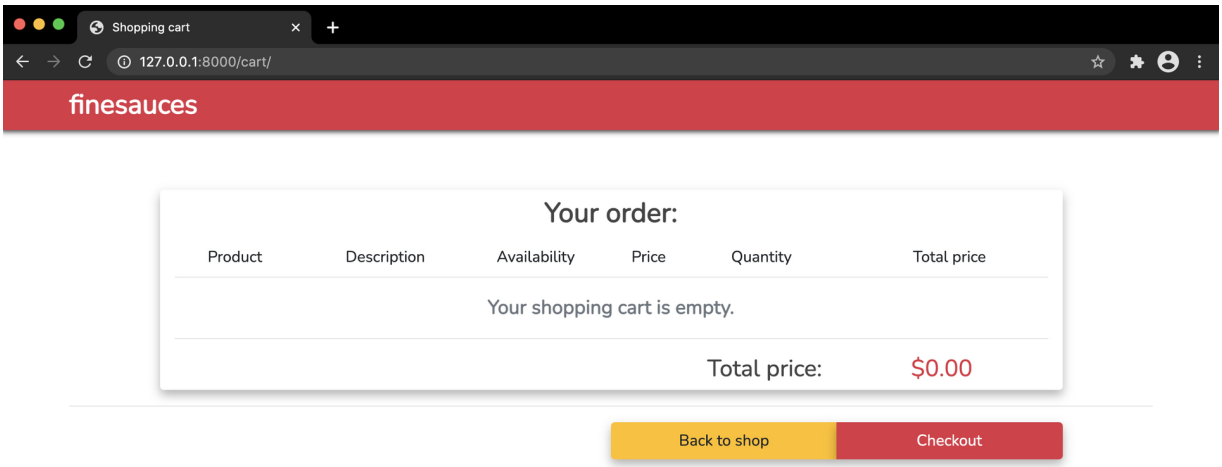
/finesauces/cart/templates/detail.html

```
#...
<form action="{% url 'cart:cart_remove' product.id %}" method="post" >
  {% csrf_token %}
  <button type="submit" class="btn" data-toggle="tooltip"
    data-placement="top" title="Remove">
    <i class="fas fa-trash-alt"></i>
    #...
```

Make sure the local development server is running with the command *python manage.py runserver* . Open the http://127.0.0.1:8000/cart/ link in your browser. You should see a similar screen to this:



The quantity field inside the form finally displays the correct number of products in the cart. Now let's try removing items from the cart. Click on the *Remove* button. Your cart should be empty, and you should see the following message:



The last update I would like to do on this page is to remove the *Checkout* button when the cart is empty. We don't want a customer to make an empty order. Let's open the cart *detail.html* template one more time and find the following code:

/finesauces/cart/templates/detail.html

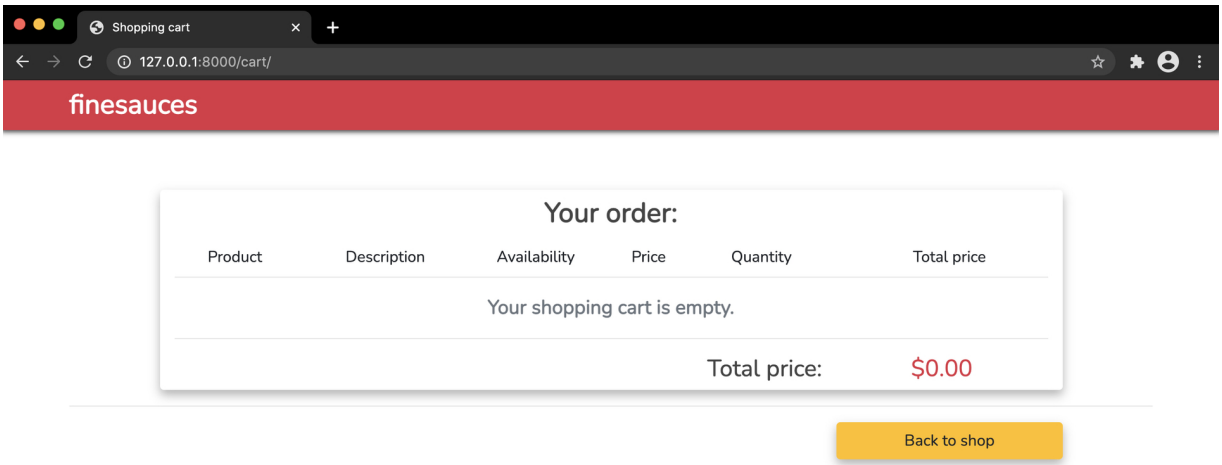
```
#...
<div class="col-lg-10 mx-auto mb-5">
  <div class="row justify-content-end">
     <div class="col-lg-6 px-0">
      <div class="btn-group d-flex">
        <a href="{% url 'listings:product_list' %}"
          class="btn btn-warning shadow-custom col">
          Back to shop
        </a>
        <a href="" class="btn btn-danger shadow-custom col">
          Checkout
        </a>
      </div>
    </div>
  </div>
</div>
#...
```

replace it by:

/finesauces/cart/templates/detail.html

```
#...
<div class="col-lg-10 mx-auto mb-5">
  <div class="row justify-content-end">
    {% if cart|length > 0 %}
      <div class="col-lg-6 px-0">
        <div class="btn-group d-flex">
          <a href="{% url 'listings:product_list' %}"
            class="btn btn-warning shadow-custom col">
            Back to shop
          </a>
          <a href="" class="btn btn-danger shadow-custom col">
            Checkout
          </a>
        </div>
      </div>
    {% else %}
      <div class="col-lg-3 px-0">
        <div class="btn-group d-flex">
          <a href="{% url 'listings:product_list' %}"
            class="btn btn-warning shadow-custom col">
            Back to shop
          </a>
        </div>
       </div>
    {% endif %}
  </div>
</div>
#...
```

To get the number of items in our cart, we use the length [33] filter provided by Django. It calls *__len__()* function on our products dictionary. In case there are no items present (length = 0), we do not show a user the *Checkout* button:



4.7 Shopping cart link

It would be pretty handy to display the shopping cart link with the product cost up on the navbar, so it is visible and accessible from all of the pages. For this functionality, we will build a context processor to include the current cart in the request context, regardless of the view processing the request.

Context processor is a Python function that takes the *HttpRequest* object as an argument and returns a dictionary that gets added to the request context. Context processors are useful, especially when we need to make something available globally to all templates [34] .

Django, by default, activates a handful of context processors upon creating a new project. They are available in the *settings.py* file under the *context_processors* option inside *TEMPLATES* settings. Django also enables the *django.template.context_processors.csrf* context processor, but it's not present in the previous list and can't be deactivated for security reasons:

/finesauces/finesauces_project/settings.py

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

4.7.1 Setting our cart into the request context

Let's create our custom context processor to set the current shopping cart to the request context. By doing that, we will be able to access our cart from every page in our project.

Context processors can reside in any location. We only need to make sure that *context_processors* list in the *settings.py* file points to the correct location. Let's go ahead and create a new file *context_processors.py* inside *finesauces_project* directory. Add the following code to it:

/finesauces/finesauces_project/context_processors.py

```
from cart.views import get_cart
from decimal import Decimal

def cart(request):
    cart = get_cart(request)
    cart_total_price = sum(Decimal(item['price']) * item['quantity']
                           for item in cart.values())

     return {
        'cart_total_price': cart_total_price
    }
```

Return to the *settings.py* file and add our context processor file to the *context_processors* list:

/finesauces/finesauces_project/settings.py

```
#...
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',
                'finesauces_project.context_processors.cart',
            ],
        },
    },
]
#...
```

The cart variable is now available to all of our templates. The cart context processor will be executed every time a template is rendered using Django's *RequestContext* . Because of this, be considerate of what you actually need to provide inside your context processors as it might affect the performance and response times of your project if you overdo it.

Let's update the *base.html* template located in *listings* application. Find the following lines in the navbar:

/finesauces/listings/templates/base.html

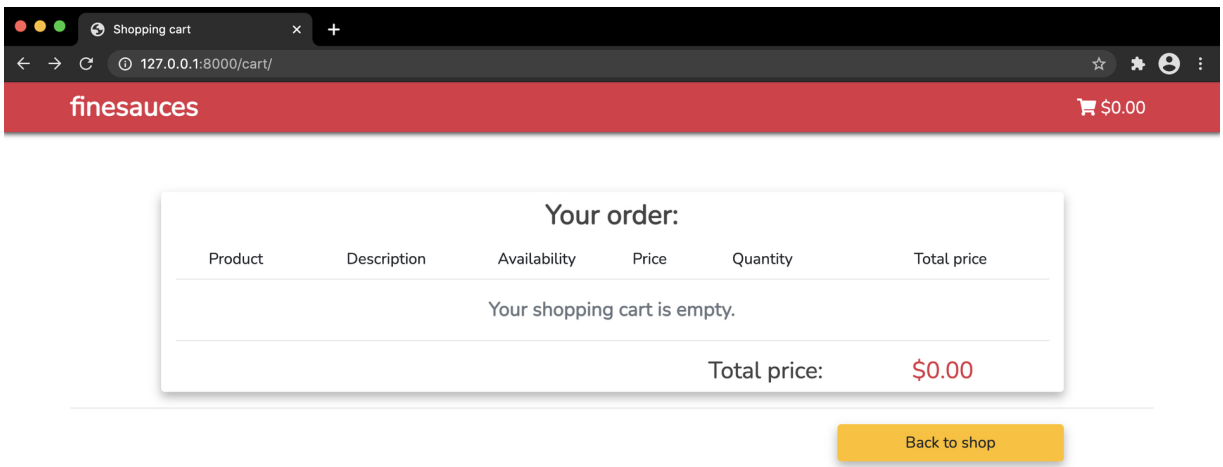
```
#...
<div class="container">
   <a href="{% url 'listings:product_list' %}" class="navbar-brand">
    <h3 class="font-weight-bold">finesauces</h3>
  </a>
</div>
#...
```

and insert the following code between ** and * * tags:

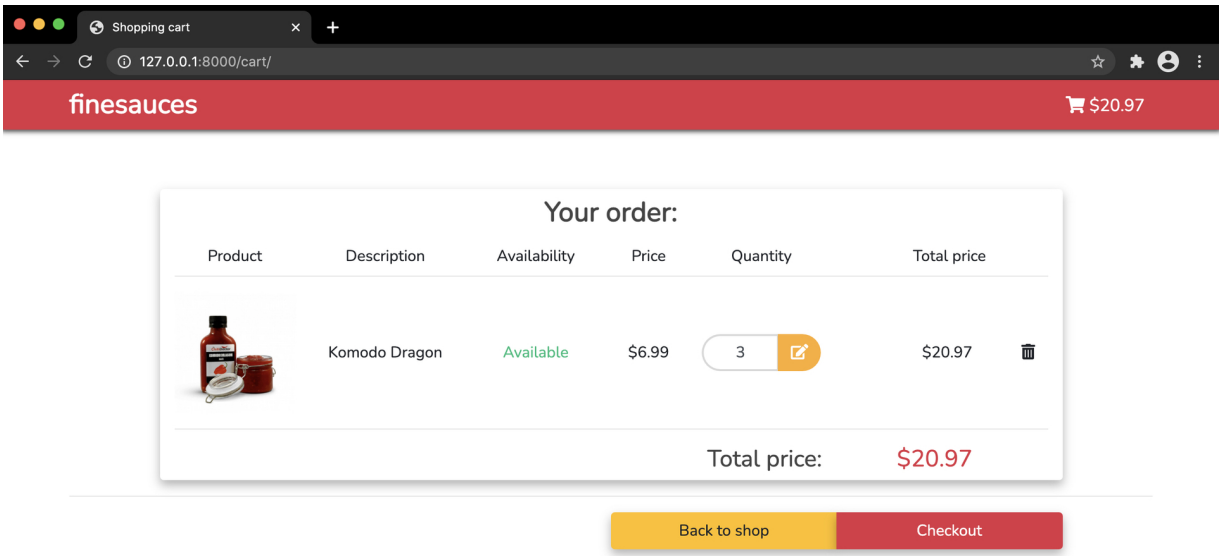
/finesauces/listings/templates/base.html

```
#...
<button class="navbar-toggler" data-toggle="collapse"
  data-target="#navbarCollapse">
  <span class="navbar-toggler-icon"></span>
</button>
<div class="collapse navbar-collapse" id="navbarCollapse">
  <ul class="navbar-nav ml-auto">
    <li class="nav-item">
      <div class="dropdown-custom">
        <a href="{% url 'cart:cart_detail' %}"
          class="nav-link text-white drop-btn">
          <i class="fas fa-shopping-cart"></i>
          $<span id="cart_price">{{ cart_total_price|floatformat:2 }}</span>
        </a>
      </div>
    </li>
  </ul>
</div>
#...
```

If we reload the cart detail page now, the cart link will appear in the top right corner of the navbar:



As expected, it will correctly display the total cost of cart items:



Perfect! We have created a fully functional shopping cart, where users can add desired products, remove them, and modify the product quantity.

Don't forget to commit the latest updates for this project and push it to the remote repository.

5 Customers orders

Customers are now able to browse through our products and add them to the shopping cart. The next step in the process will be placing an actual order and checking out the shopping cart. Order will contain contact information about the customers and products they are buying.

Let's start by creating a new application called *orders:*

terminal
```
(env) ~/finesauces$ python manage.py startapp orders
```

Let Django know about our new application by adding it to the *INSTALLED_APPS* list:

finesauces/finesauces_project/settings.py:
```
#...
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'listings.apps.ListingsConfig',
    'cart.apps.CartConfig',
    'orders.apps.OrdersConfig',
]
#...
```

5.1 Creating order models

We just activated our *orders* application. To store all order details along with the product information, we will have to create two models. Open the *models.py* file in the newly created *orders* application and add in the following code:

finesauces/orders/models.py:
```
from django.db import models
from listings.models import Product

ORDER_STATUS = [
    ('Created', 'Created'),
    ('Processing', 'Processing'),
    ('Shipped', 'Shipped'),
    ('Ready for pickup', 'Ready for pickup'),
    ('Completed', 'Completed')
]

TRANSPORT_CHOICES = [
    ('Courier delivery', 'Courier delivery'),
    ('Recipient pickup', 'Recipient pickup')
]

class Order(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField()
    telephone = models.CharField(max_length=20)
    address = models.CharField(max_length=250)
    postal_code = models.CharField(max_length=20)
    city = models.CharField(max_length=100)
    country = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=20, choices=ORDER_STATUS,
                              default='Created')
    note = models.TextField(blank=True)
    transport = models.CharField(max_length=20, choices=TRANSPORT_CHOICES)
    transport_cost = models.DecimalField(max_digits=10, decimal_places=2)

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

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

    def get_total_cost(self):
        total_cost = sum(item.get_cost() for item in self.items.all())
         total_cost += self.transport_cost
        return total_cost

class OrderItem(models.Model):
    order = models.ForeignKey(
        Order,
        related_name='items',
        on_delete=models.CASCADE
    )
    product = models.ForeignKey(
```

```
        Product,
        related_name='order_items',
        on_delete=models.CASCADE
    )
    price = models.DecimalField(max_digits=10, decimal_places=2)
    quantity = models.PositiveIntegerField()

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

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

The Order model is responsible for managing order information such as customer details, when was the order created and updated, the status of the order, optional note from customer, transport method, and related cost. For *created* and *updated* timestamps, we use slightly different arguments as they serve a different purpose. *auto_now_add* sets the field to *now* when the object is created, while *auto_now* sets the field to *now* every time the object is saved [35] . Orders will be sorted based on the *creation date* in descending order from the latest. *get_total_cost()* method calculates the total cost of the purchased items and adds related transport cost to provide total order cost.

As for the *ORDER_STATUS* and*TRANSPORT_CHOICES* lists. Remember when we implemented reviews in the *listings* application, and we defined *REVIEW_CHOICES* inside *forms.py* file? This would be another approach to specifying such options. The current approach might be a bit more compact, as you don't have to explore other files to figure out what transport options and order statuses we have at our disposal.

Run the *makemigrations* command to create initial migration for our new models:

```
terminal
(env) ~/finesauces$ python manage.py makemigrations
```

Our initial *migrations* file for *orders* application was successfully created:

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

Now we can go ahead and use these *migrations* to create tables for our models:

```
terminal
(env) ~/finesauces$ python manage.py migrate
```

Our *Order* and *OrderItem* models are successfully synced with database:

```
terminal
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, listings, orders, sessions
Running migrations:
  Applying orders.0001_initial... OK
```

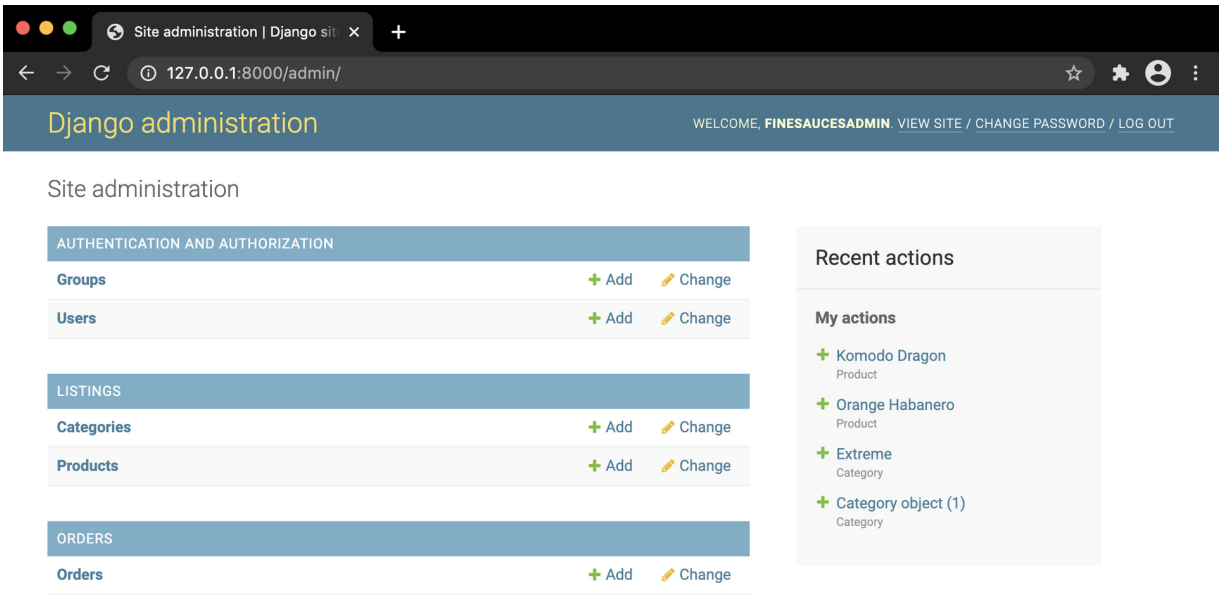
5.2 Adding models to the administration site

Now that our *Order* and *OrderItem* models are in place, we need to add them to the admin site so we are able to manage them later on. Add the following code to the *admin.py* file inside *orders* application:

```
finesauces/orders/admin.py:
from django.contrib import admin
from .models import Order, OrderItem

class OrderItemInline(admin.TabularInline):
    model = OrderItem

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

    inlines = [OrderItemInline]
```

Similar to the *Review* and *Product* admin settings, we are using a *TabularInline* subclass of *InlineModelAdmin* for the *OrderItem* model to include it as an inline in the *OrderAdmin* class, which will allow us to display *OrderItem* model on the same edit page as its related, *Order* model.

Run local development server by using *python manage.py runserver* command and visit the admin site http://127.0.0.1:8000/admin/ . Our *Order* model is now available:



If you click on the *Add* button and check the *Status* and *Transport* fields, you will notice that dropdown menus were generated dynamically based on our *ORDER_STATUS* and *TRANSPORT_CHOICES* from *models.py* file.

5.3 Creating customer order

Upon clicking on the *Checkout* button in the shopping cart, the user will be redirected to the order checkout page, which will contain an order form for filling in the required information. Once the form is submitted, *Order* and *OrderItem* objects will be created. Cart's content will be cleared by utilizing our *cart_clear()* method, and user will be redirected to the confirmation page.

Let's start by creating a form, which will be used by the customer to place an order. Create a new *forms.py* file inside the *orders* directory and add the following code:

finesauces/orders/forms.py:

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

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

        widgets = {
            'first_name': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'last_name': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'email': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'telephone': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'address': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'postal_code': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'city': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'country': forms.TextInput(
                attrs={'class': 'form-control px-2'}
            ),
            'note': forms.Textarea(
                attrs={'class': 'form-control', 'rows': 1}
            ),

            'transport': forms.RadioSelect
        }
```

Now we need to create a view that will handle our form submission and create orders accordingly. Edit the *views.py* file of the *orders* application and add the following code to it:

finesauces/orders/views.py:

```
from django.shortcuts import render, get_object_or_404
```

```
from .models import OrderItem, Order, Product
from .forms import OrderCreateForm
from cart.views import get_cart, cart_clear
from decimal import Decimal

def order_create(request):
    cart = get_cart(request)
    cart_qty = sum(item['quantity'] for item in cart.values())
    transport_cost = round((3.99 + (cart_qty // 10) * 1.5), 2)

    if request.method == 'POST':
        order_form = OrderCreateForm(request.POST)
        if order_form.is_valid():
            cf = order_form.cleaned_data
            transport = cf['transport']

            if transport == 'Recipient pickup':
                transport_cost = 0

            order = order_form.save(commit=False)
            order.transport_cost = Decimal(transport_cost)
            order.save()

            product_ids = cart.keys()
            products = Product.objects.filter(id__in=product_ids)

            for product in products:
                cart_item =cart[str(product.id)]
                OrderItem.objects.create(
                    order=order,
                    product=product,
                    price=cart_item['price'],
                    quantity=cart_item['quantity']
                )

            cart_clear(request)

            return render(
                request,
                'order_created.html',
                 {'order': order}
            )

    else:
        order_form = OrderCreateForm()

    return render(
        request,
        'order_create.html',
        {
            'cart': cart,
            'order_form': order_form,
            'transport_cost': transport_cost
        }
    )
```

We start by retrieving the cart from the session. We also get the current item count in the cart so that we can calculate delivery cost accordingly. If the customer submits the order form, the *order_create* view receives *POST* request. We instantiate this request as *order_form* and validate it. Then we retrieve the transport method from the form to see if the user requested delivery or recipient pickup. If the customer requested '*Recipient pickup*' , *transport_cost* is set to 0. By using *order = order_form.save(commit=False)* command, we create an order instance. However, by utilizing *commit=False* argument, we do not commit this order instance to the database yet. We need to set the transport cost first. After that, we create *OrderItem* object for each item in the cart. We clear the cart content and redirect the user to the order confirmation page.

For mapping this view to URL, create *urls.py* file, and add in the following code:

finesauces/orders/urls.py:

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

app_name = 'orders'

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

Don't forget to update *finesauces_project urls.py* file as well:

finesauces/finesauces_project/urls.py:

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

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

Let's add our new URL to the checkout link inside *cart detail.html* template. Open the file and find the following line:

finesauces/cart/templates/detail.html:

```
#...
<a href="" class="btn btn-danger shadow-custom col">
  Checkout
</a>
#...
```

Add our new *order_create* URL as follows:

finesauces/cart/templates/detail.html:

```
#...
<a href="{% url 'orders:order_create' %} "
  class="btn btn-danger shadow-custom col">
  Checkout
</a>
#...
```

The last piece we are missing is the templates. Create the *templates* folder within the *orders* application directory and place *order_create.html* and *order_created.html* files inside. Open *order_create.html* template and fill in the following code:

finesauces/orders/templates/order_create.html:

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

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

{% block content %}
<div class="card shadow-custom border-0 col-lg-7 mx-auto mb-3">
  <h3 class="py-2 font-weight-bold text-grey text-center">
    Delivery information:
  </h3>
  <form method="post">
    {% csrf_token %}
    <div class="row">
      <div class="col-md-6">
        <div class="input-field">
          <label class="text-muted">First name</label>
          {{ order_form.first_name }}
        </div>
      </div>
      <div class="col-md-6">
        <div class="input-field">
          <label class="text-muted">Last name</label>
          {{ order_form.last_name }}
        </div>
      </div>
      <div class="col-md-6">
        <div class="input-field">
          <label class="text-muted">Email</label>
          {{ order_form.email }}
        </div>
      </div>
      <div class="col-md-6">
        <div class="input-field">
          <label class="text-muted">Phone number</label>
          {{ order_form.telephone }}
        </div>
      </div>
    </div>
     <hr class="mt-0">
    <div class="row">
      <div class="col-md-6">
        <div class="input-field">
          <label class="text-muted">Address</label>
          {{ order_form.address }}
        </div>
      </div>
      <div class="col-md-6">
        <div class="input-field">
          <label class="text-muted">Postal code</label>
          {{ order_form.postal_code }}
        </div>
      </div>
      <div class="col-md-6">
        <div class="input-field mb-0">
          <label class="text-muted">City</label>
          {{ order_form.city }}
        </div>
      </div>
      <div class="col-md-6">
        <div class="input-field mb-0">
```

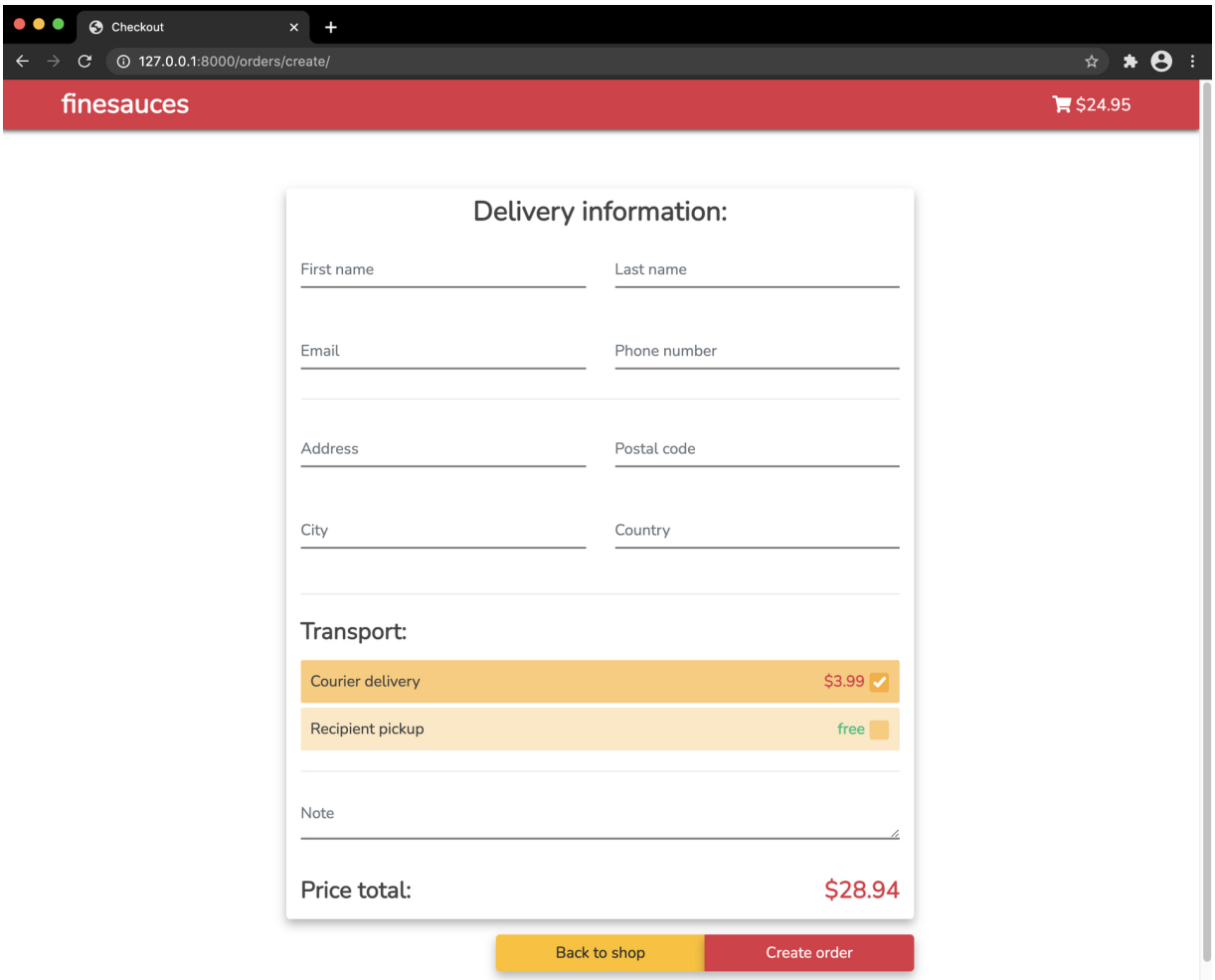
```
          <label class="text-muted">Country</label>
          {{ order_form.country }}
        </div>
      </div>
    </div>
    <hr>
    <h4 class="py-2 font-weight-bold text-grey">
      Transport:
    </h4>
    <div class="row">
      <div class="col-md-12">
        <div class="radiobtn">
          <input type="radio" id="id_transport_0" name="transport"
            value="Courier delivery" amount="{{ transport_cost }}"
            checked onclick="setTotalCost()" />
          <label for="id_transport_0">Courier delivery
            <span class="text-danger float-right pr-3 font-weight-bold">
              ${{ transport_cost }}</span>
          </label>
        </div>
         <div class="radiobtn">
          <input type="radio" id="id_transport_1" name="transport"
            value="Recipient pickup" amount="free"
            onclick="setTotalCost()" />
          <label for="id_transport_1">Recipient pickup
            <span class="text-green float-right pr-3 font-weight-bold">
              free
            </span>
          </label>
        </div>
      </div>
    </div>
    <hr class="mb-4">
    <div class="input-field">
      <label class="text-muted">Note</label>
      {{ order_form.note }}
    </div>
    <h4 class="py-2 font-weight-bold text-grey">Price total:
      <span id="order-total" class="text-danger float-right">
      </span>
    </h4>
  </div>
  <div class="col-lg-7 mx-auto mb-3">
    <div class="row justify-content-end">
      <div class="col-lg-8 px-0">
        <div class="btn-group d-flex">
          <a href="{% url 'listings:product_list' %}"
            class="btn btn-warning shadow-custom col">
            Back to shop
          </a>
          <button type="submit" class="btn btn-danger shadow-custom col">
            Create order
          </button>
        </div>
      </div>
    </div>
  </div>
</form>
{% endblock content %}
```

As for the *order_created.html* template, we can keep it simple:

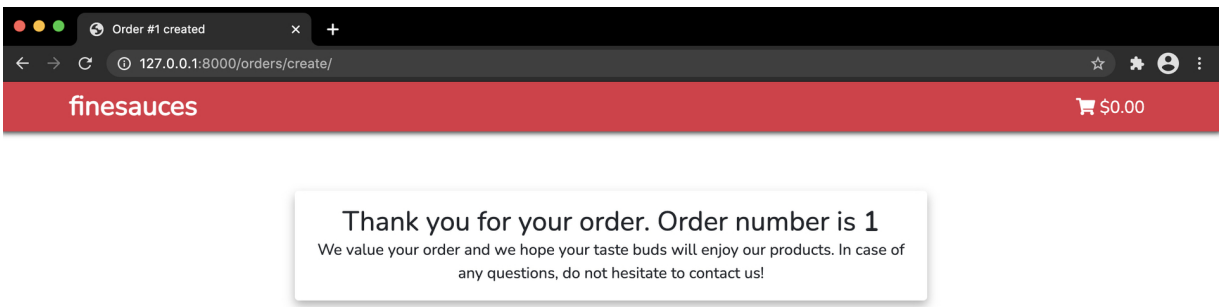
finesauces/orders/templates/order_created.html:

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

{% block title %}Order #{{ order.id }} created{% endblock %}

{% block content %}
<div class="card shadow-custom border-0 col-lg-7 mx-auto mb-3">
  <div class="py-3">
    <h3 class="text-center">Thank you for your order. Order number is
      <strong>{{ order.id }}</strong>
    </h3>
    <div class="text-center">
      We value your order and we hope your taste buds
      will enjoy our products. In case of any
      questions, do not hesitate to contact us!
    </div>
  </div>
</div>
{% endblock content %}
```

So far, in this section, we created our order form, view for processing the customer order, we did our URL mapping, and now we built order templates. Before wrapping up this part, let's go ahead and create our first order. Start your local development server with *python manage.py runserver* command, add some products to the cart, and head to the checkout page. Fill in delivery information:



and click on Create Order:



We just created our first order. And with it, the shopping cart was cleared as well. Good job so far!

5.4 Integrating Stripe payment

Our customers can add products to the cart and check out. However, there is no way for them to pay for the order and we certainly don't want to offer our products for free. We could easily provide some static payment option in the dropdown menu that users could choose from, like bank transfer or cash on delivery. It would be perfectly acceptable, but for this time and age, probably a little outdated.

That's why we will implement a card payment feature by taking advantage of the Stripe payment gateway, which is pretty straightforward and incredibly easy to use and customize.

Go ahead and visit https://stripe.com , create your account, and navigate to the dashboard at https://dashboard.stripe.com/test/developers .

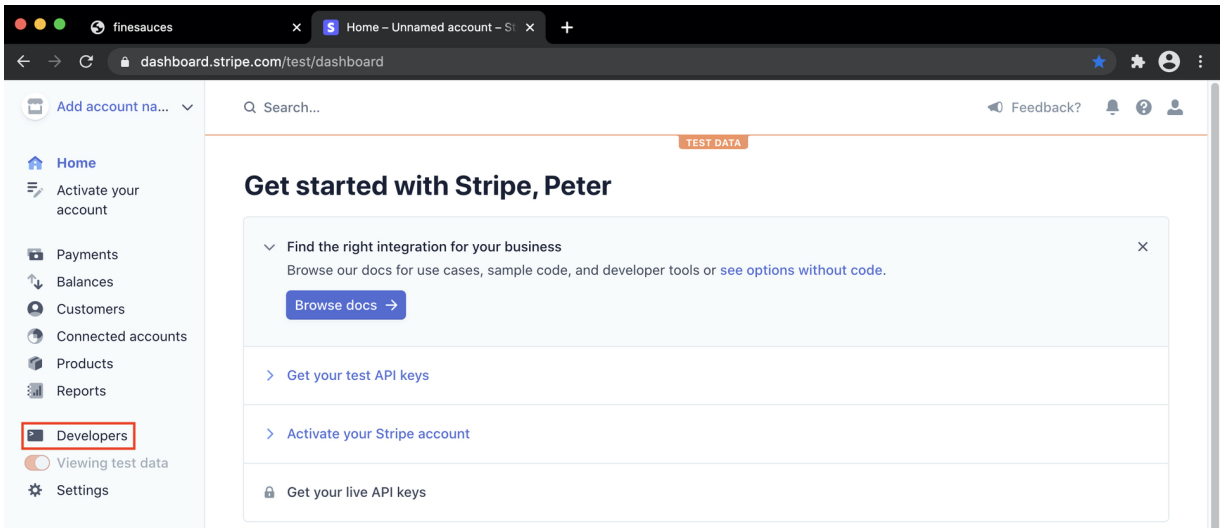
Stripe offers three ways to accept payments [36] :

- Stripe Checkout
- Charges API
- Payment Intents API

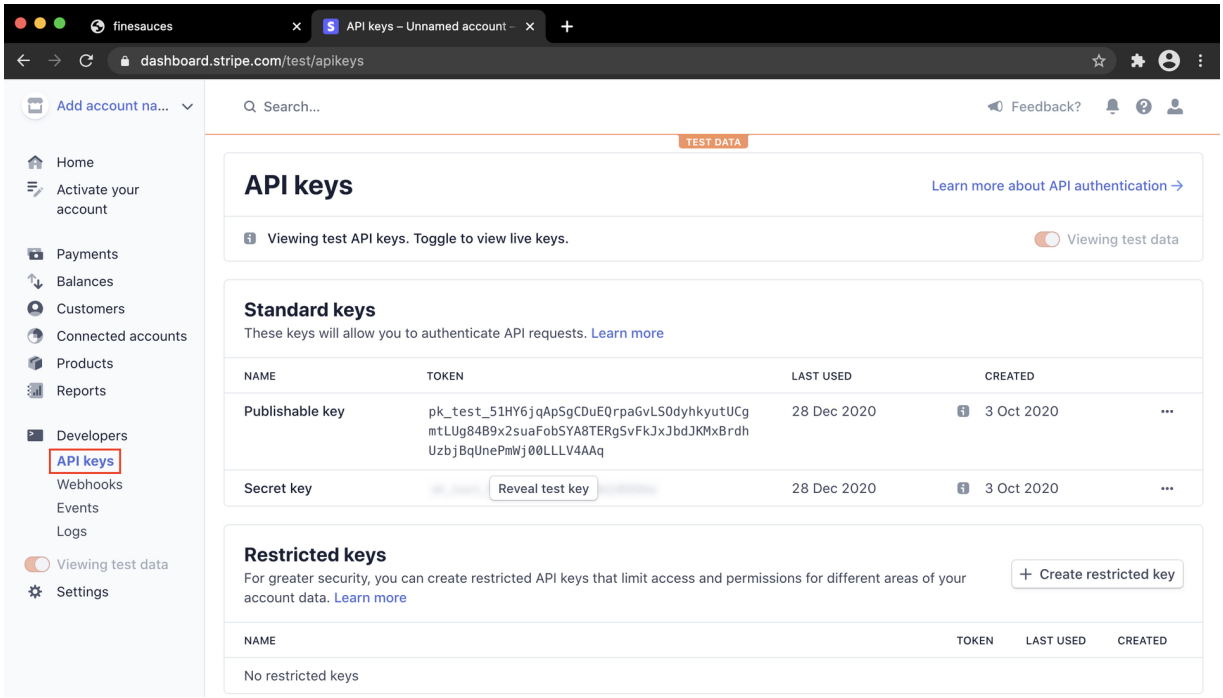
Stripe Checkout is a prebuilt payment page that you can redirect your customer to for simple purchases and subscriptions. It provides many features, such as Apple Pay, Google Pay, internationalization, and form validation.

The **Charges** and **Payment Intents APIs** enables us to build custom forms and improve user experience by matching the Stripe payment form theme with our e-commerce site. For this project, we will work with **Charges** API as it provides us with a simple way to accept customer cards.

In your dashboard, click on the *Developer* dropdown menu:



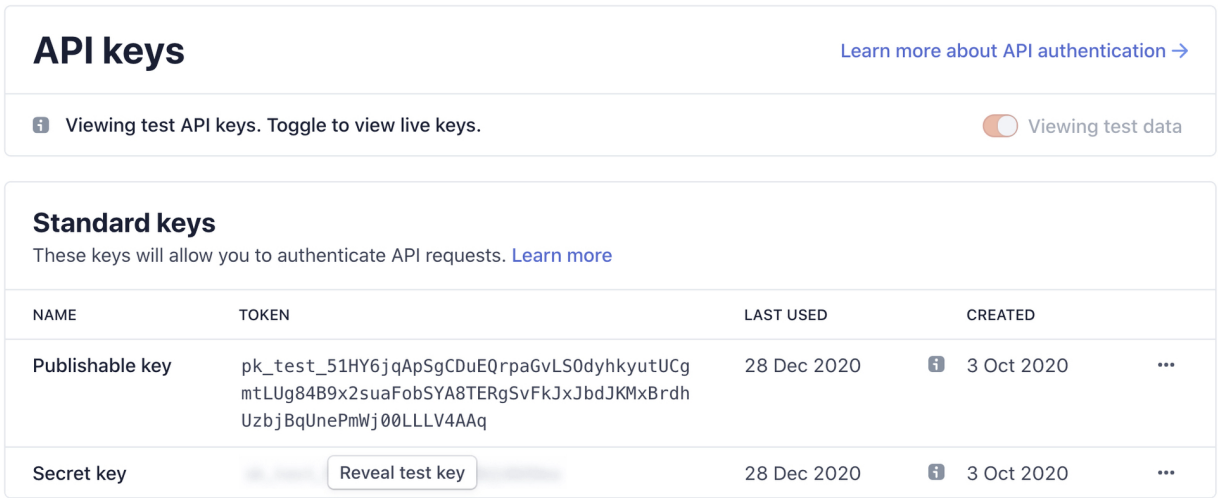
From the dropdown list, select **API keys** :



We are currently working in the test mode. It is indicated by the *TEST DATA* field on the top of the screen or in the sidebar under *Developer* section as *Viewing test data* option. The test setup is more than enough to get our payment gateway going for now.

If you, however, decide to actually start charging customers, you will need to activate live settings by filling out additional account details. For now, we will make use of dummy test cards provided by Stripe for testing.

Let's focus on our *Standard API* keys:



Stripe uses API keys to authenticate our API requests. If those API keys are not included in our requests, are incorrect or outdated, the request will not go through, and Stripe will return an error.

There are also two types of API keys: *publishable* and *secret* .

- *Publishable API* keys are meant solely to identify your account with Stripe. They aren't secret. In other words, they can safely be published on pages accessible by users.
- *Secret API* keys should be kept confidential and only stored on your servers. Account's secret API key can perform any API request to Stripe without restriction.

Let's add Stripe API keys to our Django project. Open *local_settings.py* file located in *finesauces_project* directory and add those keys to the bottom of the file:

finesauces/finesauces_project/local_settings.py:

```
#...
STRIPE_TEST_PUBLISHABLE_KEY='<your_test_publishable_key>'
STRIPE_TEST_SECRET_KEY='<your_test_secret_key>'
```

Fill in your keys here. Make sure to never disclose your secret key anywhere!

5.4.1 Adding Stripe to our view

Let's start by getting a Python library for Stripe:

terminal

```
(env) ~/finesauces$ pip install stripe
```

Now go ahead and open the *views.py* file located in the *orders* application and add the following imports to it:

finesauces/orders/views.py

```
#...
from django.conf import settings
import stripe

stripe.api_key = settings.STRIPE_TEST_SECRET_KEY
#...
```

We import *stripe* library and *settings* configuration so we can reference our *stripe secret key* in *order_create* view. We will charge the customer right after our order, and order items are processed. Find the following code:

finesauces/orders/views.py

```
#...
for product in products:
    cart_item =cart[str(product.id)]
    OrderItem.objects.create(
        order=order,
        product=product,
        price=cart_item['price'],
        quantity=cart_item['quantity']
    )

cart_clear(request)
#...
```

and insert the following code between the *for* loop and *cart.clear()* as follows:

finesauces/orders/views.py

```
#...
for product in products:
    cart_item =cart[str(product.id)]
    OrderItem.objects.create(
        order=order,
        product=product,
        price=cart_item['price'],
        quantity=cart_item['quantity']
    )

customer = stripe.Customer.create(
    email = cf['email'],
    source = request.POST['stripeToken']
)

charge = stripe.Charge.create (
    customer = customer,
    amount = int(order.get_total_cost() * 100),
    currency='usd',
    description = order
)

cart_clear(request)
#...
```

We first create a *customer* object which will be associated with the Stripe charge. This object is created with the following fields:

- ***email*** - email address of the customer. This field is optional, and you can specify a lot more attributes, such as *address* , *description* , *name* , *phone* , *shipping,* and many more. Check https://stripe.com/docs/api/customers/create

for more information on the *Customer* object.
- ***source*** - tokenized representation of the payment card. When a customer submits the payment form, card data is sent to the Stripe server. We then receive this token, which we then pass to our view.

In *charge* object, we specify:

- ***customer*** - customer object we just created. It helps us associate the payment with a specific customer
- ***amount*** - positive integer value in the smallest currency unit (100 cents to charge $1.00). We retrieve total order cost as decimal, multiply it by 100 to get total cost in cents, and then convert it to integer
- ***currency*** - three-letter ISO currency code in lowercase. For a complete list of supported currencies, visit https://stripe.com/docs/currencies .
- ***description*** - we specify this field to indicate which order was paid. In *Order* model, we defined *__str__()* function to return *f'Order #{self.id}'* string, meaning the description field would be set to *Order #1* , if we made the charge for our first order.

5.4.2 Using Stripe elements

Stripe Elements is a set of pre-built UI components, such as inputs and buttons, for building checkout flow. It is provided as a part of *Stripe.js* . Elements include convenient features such as formatting card information automatically as the user types it in, using responsive design to fit our customer's screen or mobile device's width, and enables us to style the form to match our e-commerce checkout form. Stripe Elements documentation is available at https://stripe.com/docs/stripe-js#elements .

It also comes with all necessary components (HTML, CSS, Javascript) for styling and implementing our payment form. Let's start with the *HTML* code. Open up the Elements documentation page and look for the following section:

Stripe Elements

Stripe Elements is a set of prebuilt UI components, like inputs and buttons, for building your checkout flow. It's available as a feature of Stripe.js. Stripe.js tokenizes the sensitive information within an Element without ever having it touch your server.

Elements includes features like:

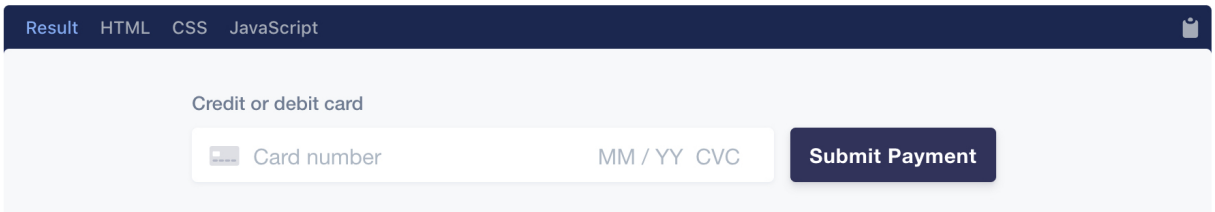
- Formatting card information automatically as it's entered
- Translating placeholders into your customer's preferred language
- Using responsive design to fit the width of your customer's screen or mobile device
- Customizing the styling to match the look and feel of your checkout flow

STRIPE CHECKOUT

If you'd rather not build your own payment form, consider Checkout, a Stripe-hosted page to accept payments for one-time purchases and subscriptions.

HTML + JS | React

The Card Element lets you collect card information all within one Element. It includes a dynamically-updating card brand icon as well as inputs for number, expiry, CVC, and postal code. Get started with accepting a payment.



Select *HTML* tab:

Result HTML CSS JavaScript

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

<form action="/charge" method="post" id="payment-form">
  <div class="form-row">
    <label for="card-element">
      Credit or debit card
    </label>
    <div id="card-element">
      <!-- A Stripe Element will be inserted here. -->
    </div>

    <!-- Used to display form errors. -->
    <div id="card-errors" role="alert"></div>
  </div>

  <button>Submit Payment</button>
</form>
```

We will use most of the code provided here, but we need to split it a little due to our custom checkout form structure. Open the *order_create.html* template and find the following section:

finesauces/orders/templates/order_create.html

```
#...
<h3 class="py-2 font-weight-bold text-grey text-center">
  Delivery information:
</h3>
<form method="post">
  {% csrf_token %}
  #...
```

Copy the *<script src="https://js.stripe.com/v3/"></script>* element and add it before opening *<form>* tag. Then update *<form>* tag to include *id="payment-form"* . After these steps you should end up with:

finesauces/orders/templates/order_create.html

```
#...
<h3 class="py-2 font-weight-bold text-grey text-center">
  Delivery information:
</h3>

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

<form method="post" id="payment-form">
  {% csrf_token %}
  #...
```

Now scroll down below the *Transport* section and above the *Note* field and locate this code:

finesauces/orders/templates/order_create.html

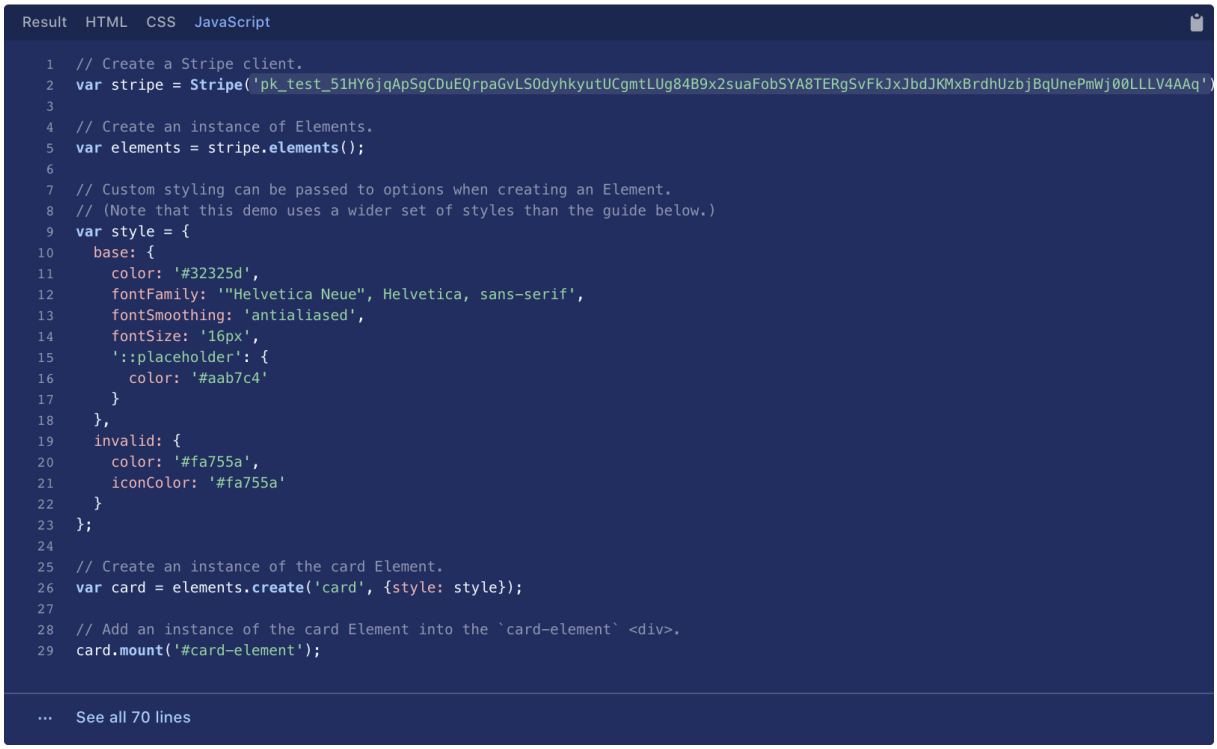
```
#...
</div>
<hr class="mb-4">
<div class="input-field">
  <label class="text-muted">Note</label>
  {{ order_form.note }}
</div>
#...
```

Insert the highlighted code from the *HTML* Elements example along with our custom h4 header between * * and *<hr>* tags:

finesauces/orders/templates/order_create.html

```
#...
</div>
<h4 class="py-2 font-weight-bold text-grey">Payment:</h4>
<div class="form-row">
  <div id="card-element">
  <!-- A Stripe Element will be inserted here. -->
  </div>

  <!-- Used to display form errors. -->
  <div id="card-errors" role="alert"></div>
</div>
<hr class="mb-4">
<div class="input-field">
  <label class="text-muted">Note</label>
  {{ order_form.note }}
</div>
#...
```

We are also interested in the *Javascript* tab, so go ahead and select it:



Create *<script></script>* tags between *</form>* and *{% endblock content %}* at the bottom of the file and paste in the code from *Javascript* tab between those *<script></script>* tags:

finesauces/orders/templates/order_create.html

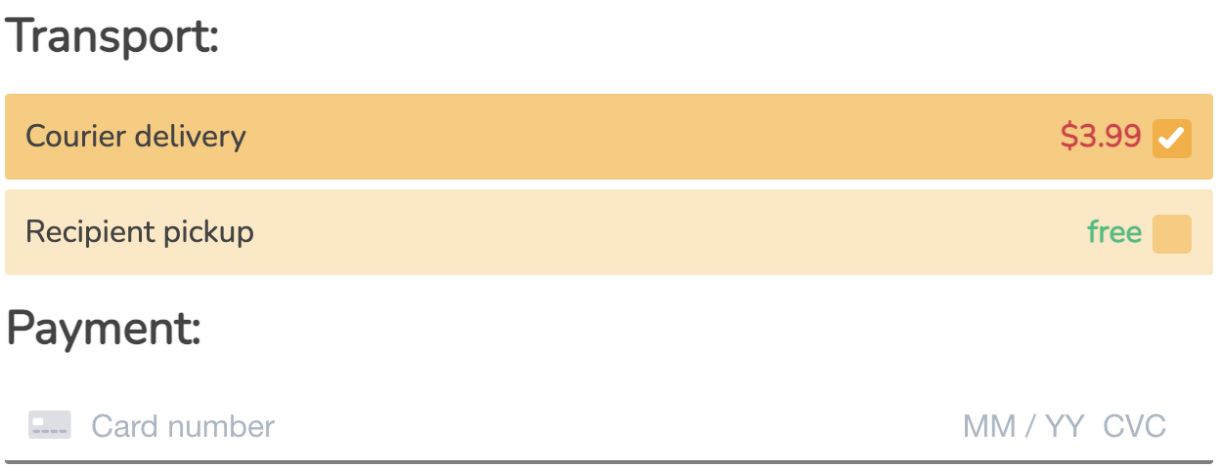
```
#...
</form>
<script>
  // Create a Stripe client.
                                                var                          stripe                          =
Stripe('pk_test_51HY6jqApSgCDuEQrpaGvLSOdyhkyutUCgmtLUg84B9x2suaFobSYA8TERgSvFkJxJbdJKMxBrdhUzbj
BqUnePmWj00LLLV4AAq');

  // Create an instance of Elements.
  var elements = stripe.elements();

  // Custom styling can be passed to options when creating an Element.
  // (Note that this demo uses a wider set of styles than the guide below.)
  var style = {
    base: {
      color: '#32325d',
      fontFamily: '"Helvetica Neue", Helvetica, sans-serif',
      fontSmoothing: 'antialiased',
      fontSize: '16px',
      '::placeholder': {
        color: '#aab7c4'
      }
    },
    invalid: {
      color: '#fa755a',
      iconColor: '#fa755a'
    }
  };

  // Create an instance of the card Element.
  var card = elements.create('card', { style: style });

  // Add an instance of the card Element into the `card-element` <div>.
  card.mount('#card-element');
  // Handle real-time validation errors from the card Element.
  card.on('change', function (event) {
    var displayError = document.getElementById('card-errors');
    if (event.error) {
      displayError.textContent = event.error.message;
     } else {
      displayError.textContent = '';
    }
  });

  // Handle form submission.
  var form = document.getElementById('payment-form');
  form.addEventListener('submit', function (event) {
    event.preventDefault();

    stripe.createToken(card).then(function (result) {
      if (result.error) {
        // Inform the user if there was an error.
        var errorElement = document.getElementById('card-errors');
        errorElement.textContent = result.error.message;
```

```
      } else {
        // Send the token to your server.
        stripeTokenHandler(result.token);
      }
    });
  });

  // Submit the form with the token ID.
  function stripeTokenHandler(token) {
    // Insert the token ID into the form so it gets submitted to the server
    var form = document.getElementById('payment-form');
    var hiddenInput = document.createElement('input');
    hiddenInput.setAttribute('type', 'hidden');
    hiddenInput.setAttribute('name', 'stripeToken');
    hiddenInput.setAttribute('value', token.id);
    form.appendChild(hiddenInput);

    // Submit the form
    form.submit();
  }
</script>
{% endblock content %}
#...
```

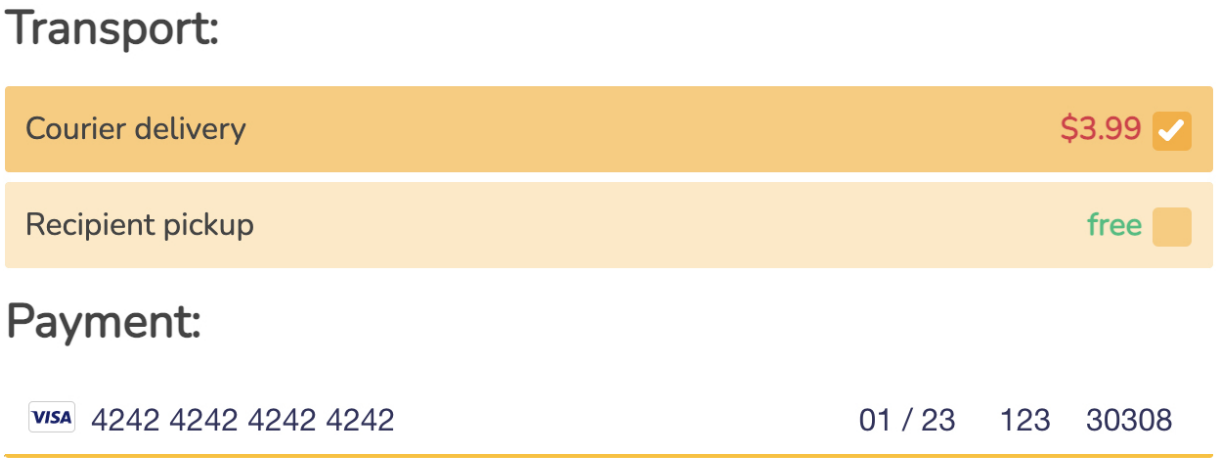
Important thing to remember! *var stripe = Stripe('...')* will be unique for each of the Stripe accounts. In this case, it is my account's *STRIPE_TEST_PUBLISHABLE_KEY* . Stripe is kind enough to fill that code in dynamically for each account. Therefore, your key will be different. As for the form styling, we do not need to utilize the *CSS* tab as our custom styling is defined in the *style.css* file right at the end.

5.4.3 Placing an order

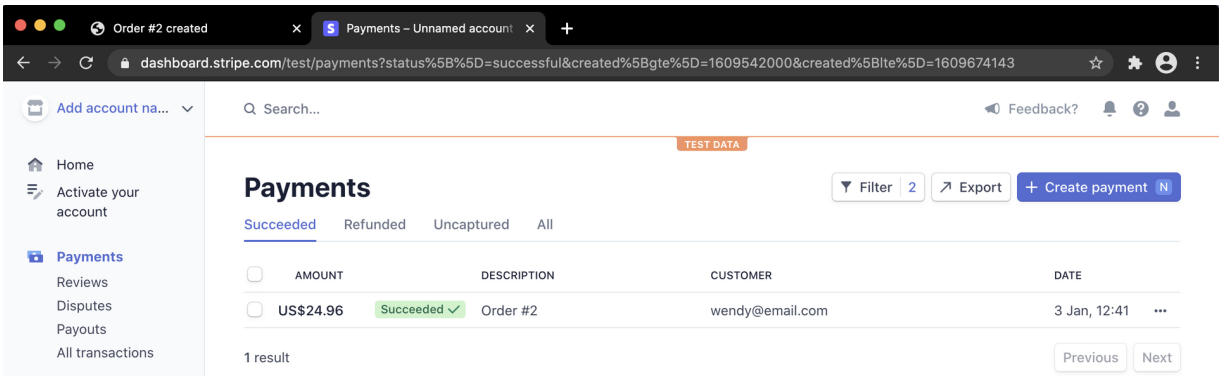
Let's have a look at our custom Stripe payment form. Make sure your development server is running by using the *python manage.py runserver* command. Fill your shopping cart with products and head to the *Checkout* form. You should see a new addition to our form right below the *Transport* field:



Go ahead and fill in the required Delivery Information. For the card information, Stripe provides us with a handful of test cards [37] . Make sure to set the expiry date in the future. You can use any random CVC number. Our card form handles card validation by default.



After checking out the order, we will be redirected to the familiar confirmation page. Let's confirm that our payment went through successfully. We can verify that on the Stripe page within the *Payments* [38] section:



You can see that the *DESCRIPTION* field of the order is set to Order #2, as we specified in the *Order* model. As for the customer, we can see the email information, which we previously specified in our view.

5.5 Sending email notifications

After checking out the order, the customer will probably expect some sort of email notification regarding his purchase. If we were to send that notification directly from our view using *Simple Mail Transfer Protocol (SMTP)*, it would most probably slow down our site response time or could cause issues in case of a temporary connection outage. That's where asynchronous tasks come in.

5.5.1 Setting up Celery

Celery is a simple, flexible, and reliable distributed system to process vast amounts of messages while providing operations with the tools required to maintain such a system. It's a task queue with a focus on real-time processing while also supporting task scheduling [39] .

Celery is available as a Python package. That means we can go ahead and install it using *pip* :

terminal
```
(env) ~/finesauces$ pip install "celery==4.*"
```

To use Celery in our project, we need to define an instance of the Celery library called an *app*. The recommended way is to create a *celery.py* module within our main project directory *(finesauces_project)* . Let's create this file and add in the following code:

finesauces/finesauces_project/celery.py
```
import os
from celery import Celery

# set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'finesauces_project.settings')

app = Celery('finesauces_project')

# Using a string here means the worker doesn't have to serialize
# the configuration object to child processes.
# - namespace='CELERY' means all celery-related configuration keys
#   should have a `CELERY_` prefix.
app.config_from_object('django.conf:settings', namespace='CELERY')

# Load task modules from all registered Django app configs.
app.autodiscover_tasks()
```

First, we set the default *DJANGO_SETTINGS_MODULE* environment variable for the *celery* command-line program. We could omit it, but it will save us from always passing in the settings module to the celery program. It must always come before creating an *app* instance, which we create next.

Taken from Celery documentation [40] , adding the Django settings module as a configuration source for Celery enables us to configure Celery directly from the Django settings. The uppercase namespace means that all Celery configuration options must be specified in uppercase instead of lowercase, and start with *CELERY_* , so for example, the *task_always_eager* setting becomes *CELERY_TASK_ALWAYS_EAGER* , and the *broker_url* setting needs to be defined as *CELERY_BROKER_URL* . This also applies to the worker's settings, where the *worker_concurrency* setting becomes *CELERY_WORKER_CONCURRENCY* , for example:

```
# Celery Configuration Options
CELERY_TIMEZONE = "Australia/Tasmania"
CELERY_TASK_TRACK_STARTED = True
CELERY_TASK_TIME_LIMIT = 30 * 60
```

By using the command:

```
app.autodiscover_tasks()
```

we tell Django to go through our applications and discover *tasks.py* modules.

To ensure that our Celery app is loaded when Django starts, we need to import this app in our *finesauces_project/__init__.py* module:

finesauces/finesauces_project/__init__.py
```
from .celery import app as celery_app
```

Celery requires a solution to send and receive messages; usually, this comes in the form of a separate service called a *message broker* . There are a couple of brokers to choose from. The most popular and supported options would probably be *RabbitMQ* [41] and *Redis* [42] . We will opt for *RabbitMQ* in this project.

5.5.2 Setting up RabbitMQ

RabbitMQ is a lightweight and easy to deploy message broker. It supports multiple messaging protocols. It can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements.

If you are using Debian or Ubuntu based distribution, run the command:

terminal
```
(env) ~/finesauces$ sudo apt install rabbitmq-server
```

For macOS, run:

terminal
```
(env) ~/finesauces$ brew install rabbitmq
```

For other platforms' installation instructions, please reviews documentation available at https://www.rabbitmq.com/download.html .

To confirm everything was installed properly, open a new terminal window and start the RabbitMQ server by using the command:

terminal
```
(env) ~/finesauces$ sudo rabbitmq-server
```

The terminal output should look similar to this:

terminal
```
  ##  ##      RabbitMQ 3.8.9
  ##  ##
  ##########  Copyright (c) 2007-2020 VMware, Inc. or its affiliates.
  ######  ##
  ##########  Licensed under the MPL 2.0. Website: https://rabbitmq.com

   Doc guides: https://rabbitmq.com/documentation.html
  Support:    https://rabbitmq.com/contact.html
  Tutorials:  https://rabbitmq.com/getstarted.html
  Monitoring: https://rabbitmq.com/monitoring.html

  Logs: /usr/local/var/log/rabbitmq/rabbit@localhost.log
        /usr/local/var/log/rabbitmq/rabbit@localhost_upgrade.log

  Config file(s): (none)

  Starting broker... completed with 6 plugins.
```

5.5.3 Adding asynchronous tasks to our application

Let's create our asynchronous task to send an email notification to users when they place an order. A convention is to include asynchronous tasks for our applications in a task module within our application directory.

Create a new file inside the *orders* application directory and name it *tasks.py* . As we already established, this is the place where Celery will look for asynchronous tasks. Add the following code to it:

finesauces/orders/tasks.py
```
from celery import task
from django.core.mail import send_mail
from .models import Order

@task
def order_created(order_id):
    order = Order.objects.get(id=order_id)
    subject = f'Order nr. {order.id}'
    message = f'Dear {order.first_name}, \n\n' \
              f'Your order was successfully created.\n' \
              f'Your order ID is {order.id}.'
    mail_sent = send_mail(
        subject,
        message,
        'eshop@finesauces.store',
        [order.email]
    )

     return mail_sent
```

Here we define our *order_created* task by using the task decorator. As you can see, a Celery task is just a Python function decorated with @*task* . Our task function receives an *order_id* parameter. It's always recommended to only pass IDs to task

functions and lookup objects when the task is executed. We use the *send_mail()* function provided by Django to send an email notification to the user who placed an order. In the *send_mail()* we specify the subject of the email, message, sender, and recipient emails.

Now let's add this task to our *order_create* view. Edit the *views.py* file of the *orders* application and import our *order_created* task:

finesauces/orders/views.py
```
from .tasks import order_created
#...
```

Then add the *order_created* asynchronous task to the *order_create* view as follows:

finesauces/orders/views.py
```
#...
cart_clear(request)

order_created.delay(order.id)
#...
```

We call the *delay()* [43] method of the task to execute it asynchronously. This task will be added to the queue and will be executed by the worker.

5.5.4 Simple Mail Transfer Protocol (SMTP) setup

The last thing we need to set up is a connection to the SMTP server. We can use a local *Simple Mail Transfer Protocol (SMTP)* server, or we can specify a connection to the external one if we have access to any.

If we decide to use a local SMTP server, Django will print the message to the terminal instead of actually sending it. Let's try it out. Go ahead and open the *settings.py* file located in the *finesauces_project* directory. Add the following line to the end of the file:

finesauces/finesauces_project/settings.py
```
#...
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Open another terminal and start the Celery worker from your main *finesauces* directory using the following command:

terminal
```
(env) ~/finesauces$ celery -A finesauces_project worker -l info
```

Let's create a new order. Add some products to the shopping cart, fill in the required information, and checkout. You should see a similar message in your terminal:

terminal
```
[2021-01-03 12:02:44,506: INFO/MainProcess] Received task:
orders.tasks.order_created[b06f7a7f-4406-4feb-aa4f-d88688e390c0]
[2021-01-03 12:02:44,533: WARNING/ForkPoolWorker-8] Content-Type:
text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Order nr. 3
From: eshop@finesauces.store
To: vanessa@email.com
Date: Sun, 03 Jan 2021 12:02:44 -0000
Message-ID: <160967536452.2744.12570738496509036033@192.168.1.14>

Dear Vanessa,

Your order was successfully created.
Your order ID is 3.
[2021-01-03 12:02:44,534: WARNING/ForkPoolWorker-8] ----------------------------------
[2021-01-03 12:02:44,534: INFO/ForkPoolWorker-8] Task
orders.tasks.order_created[b06f7a7f-4406-4feb-aa4f-d88688e390c0]
succeeded in 0.026656446999995856s: 1
```

If you would like to use services of external SMTP server and send email from your custom ecommerce domain, you will need to fill in and add the following configuration to the *local_settings.py* file (to keep them secret):

finesauces/finesauces_project/local_settings.py
```
#...
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST =
EMAIL_HOST_USER =
EMAIL_HOST_PASSWORD =
EMAIL_PORT =
EMAIL_USE_SSL =
DEFAULT_FROM_EMAIL = EMAIL_HOST_USER
```

To get started, you could most probably use the SMTP server of your email service provider, such as *Gmail* [44] or *Outlook* [45] . Other very popular options are *MailGun* [46] and *Mailchimp* [47] .Let's push our project to the remote repository and continue.

6 Extending administration site

One day, when your e-commerce business takes off, you might be interested in doing a little bit of exporting, reporting, and analysis. It would be nice to be able to export your data to some kind of a report. One of the most popular formats would be .xlsx. Django admin site, by default, does not offer any functionality for this use case. That's why we are going to build it ourselves.

6.1 Adding custom actions to the administration site

We are going to modify the object list view to include our custom administration action. We can implement custom admin action that will allow staff members to apply specific actions to multiple records at the same time.

Let's have a look at how this works. Make sure your local development server is up and running and access http://127.0.0.1:8000/admin/orders/order/ , where you will be able to see the list of all orders created so far.

A staff member can select multiple records at the same time and apply the desired action. For now, we are only able to delete selected records. Let's change that.

Django action is basically a standard function that accepts three parameters:

- ***modeladmin*** - represents the model that is currently being displayed. In our case, that would be the *Order* model
- ***request*** - the current request object as an HttpRequest instance
- ***queryset*** - QuerySet for the objects that are selected

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

6.1.1 Exporting data into .xlsx report

Before getting to our custom admin action, we need to install a Python library for handling our .xlsx files. Run the following command to install *xlsxwriter:*

terminal
```
(env) ~/finesauces$ pip install xlsxwriter
```

Now open the *admin.py* file located in the *orders* application directory and add the following code before *OrderAdmin* class:

finesauces/orders/admin.py
```
from django.contrib import admin
from .models import Order, OrderItem, Product
import xlsxwriter
import datetime
from django.http import HttpResponse

def export_to_xlsx(modeladmin, request, queryset):
    opts = modeladmin.model._meta
    dateTimeObj = datetime.datetime.now()
    timestamp = dateTimeObj.strftime("%d-%b-%Y")

    content_disposition = f'attachment; filename=orders_{timestamp}.xlsx'
    response = HttpResponse(content_type=
        'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet')
    response['Content-Disposition'] = content_disposition
    workbook = xlsxwriter.Workbook(response, {'in_memory': True})
    worksheet = workbook.add_worksheet()

    fields = [field for field in opts.get_fields() \
        if not field.one_to_many]
    header_list = [field.name for field in fields]

    products = Product.objects.all()
    for product in products:
        header_list.append(f'{product.name} qty')
        header_list.append(f'{product.name} price')

    header_list.append('order_price_total')

     # write the header
    for column, item in enumerate(header_list):
        worksheet.write(0, column, item)

    # write data rows
    for row, obj in enumerate(queryset):

        # load the order item details into dictionary
        prod_tracker = {product.name:{'qty': 0, 'price': 0} \
            for product in products}

        order_items = obj.items.all()
        for item in order_items:
            prod_tracker[item.product.name]['qty'] = item.quantity
            prod_tracker[item.product.name]['price'] = item.price

        data_row = []
        order_price_total = 0

        # iterate through fields in Order object
```

```
        # and append data to data_row list
        for field in fields:
            value = getattr(obj, field.name)
            if field.name == 'transport_cost':
                order_price_total += value
            if isinstance(value, datetime.datetime):
                value = value.strftime('%d/%m/%Y')
            data_row.append(value)

        # iterate through order item data - qty and price
        # and append to data_row list
        for product in prod_tracker:
            for qty_price in prod_tracker[product]:
                data_row.append(prod_tracker[product][qty_price])
            order_price_total += (
                prod_tracker[product]['qty'] * prod_tracker[product]['price']
            )

        # append total cost to data_row list
        data_row.append(order_price_total)

        for column, item in enumerate(data_row):
             worksheet.write(row + 1, column, item)

    workbook.close()

    return response

export_to_xlsx.short_description = 'Export to XLSX'
#...
```

Let's break down this function:

- we start by instantiating Model *_meta* API. This is core Django ORM API, that will enable us to access all fields of our Order model

- then we create current date timestamp. We will append it to the name of our report

- we define *content_disposition* to indicate, that HttpResponse will contain attachment, which will be our .xlsx report

- then we declare that the response containing our report should be treated accordingly to the file format

- then we create workbook and worksheet

-a fter that we utilize our _meta API and create fields list. We exclude *field.one_to_many,* which represents *Order* reference to *OrderItems* , since we don't want to include that field in the report – the final list*(fields)* looks like this:

[<django.db.models.fields.AutoField: id>,
<django.db.models.fields.CharField: first_name>,
<django.db.models.fields.CharField: last_name>,
<django.db.models.fields.EmailField: email>,
<django.db.models.fields.CharField: telephone>,
<django.db.models.fields.CharField: address>,
<django.db.models.fields.CharField: postal_code>,
<django.db.models.fields.CharField: city>,
<django.db.models.fields.CharField: country>,
<django.db.models.fields.DateTimeField: created>,
<django.db.models.fields.DateTimeField: updated>,
<django.db.models.fields.CharField: status>,
<django.db.models.fields.TextField: note>,
<django.db.models.fields.CharField: transport>,
<django.db.models.fields.DecimalField: transport_cost>]

- then we build *header_list* , which will contain names of the fields. We will use that list to create header for our report. We will append couple of other fields to it soon. For now, it looks as such:

['id', 'first_name', 'last_name', 'email', 'telephone', 'address', 'postal_code', 'city', 'country', 'created', 'updated', 'status', 'note', 'transport', 'transport_cost']

- we retrieve the *QuerySet* containing all products on our ecommerce site. We iterate through it and for each product, we append *f'{product.name} qty'* and *f'{product.name} qty'* to the header column list. We also want to know the total cost of given order, so we also append *'order_price_total'* to the header list. We then write the header to the worksheet

- then we start to iterate through the *QuerySet* containing all of our *Orders* . In the export, we will have columns for every product. If some of the products were not ordered in given order, we want to set the value to 0 for quantity and price. Let's say if we have 2 products in total, but customer ordered only one, the other will have its quantity and price set to 0. As we go through our *QuerySet* , we create dictionary for every *Order* . Each of the products is represented as a key and it has another nested dictionary inside, where quantity and price are set to 0.

- after that, we go through ordered items for a given order and mark the quantity and price in our dictionary.

- we define *the data_row list, which will hold values for the current order, and order_price_total, which will track the order's cost* .

- based on our *fields* list, we retrieve the attribute from the current order and append it to the *data_row* , which holds values for the current order.

- we then go through our product tracker dictionary, calculate the total price per product, and add it to the *order_price_total* .

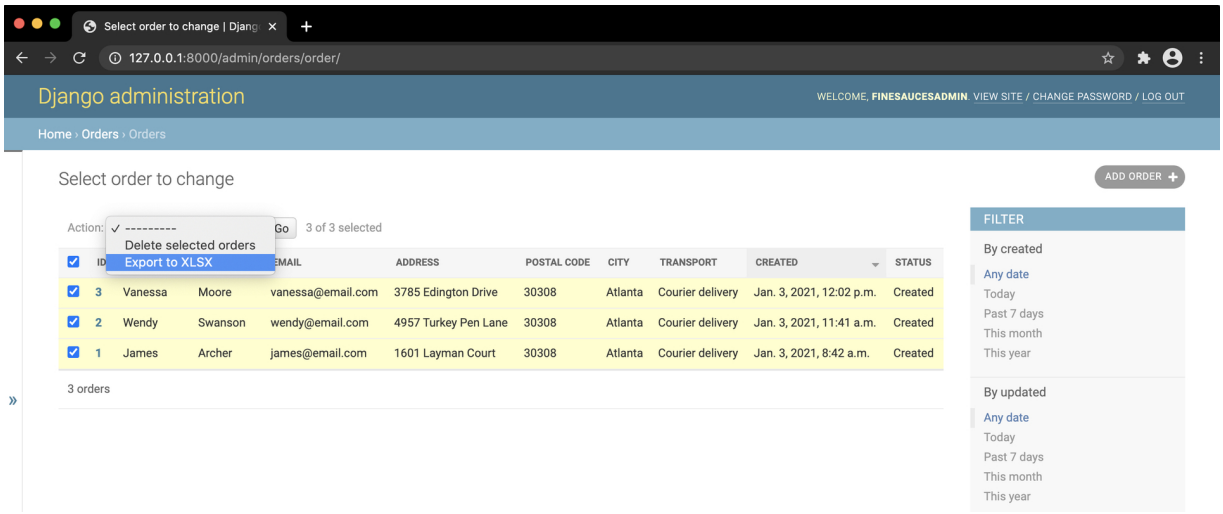
We configured the display name for the action in the *actions* dropdown element of the administration site by setting a *short_description* attribute on the function

Don't forget to add *export_to_xlsx* action to the *OrderAdmin* class:

finesauces/orders/admin.py

```
#...
@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    #...
    inlines = [OrderItemInline]

    actions = [export_to_xlsx]
```

We have just created our first custom Django admin action. Let's have a look inside our admin site and test it out! Start the development server by using the *runserver* command and access http://127.0.0.1:8000/admin/orders/order/ . Go ahead and select a couple of records, pick *Export to XLSX* action from the dropdown menu, and click *Go* to create the report:



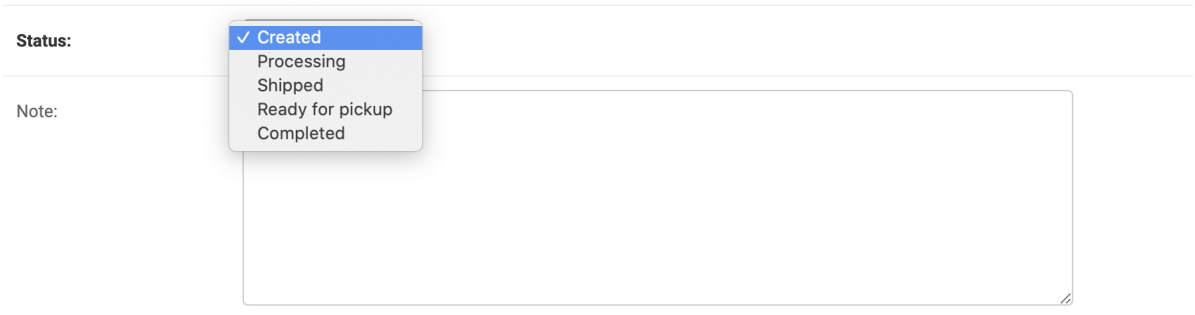
Report with current date will be generated for us:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	id	first_name	last_name	email	telephone	address	postal_cod	city	country	created	updated	status	note	transport	transport_(	Orange Hat	Orange Hat	Komodo Di	Komodo Di	order_price_total	
2	3	Vanessa	Moore	vanessa@e	222-555-0	3785 Eding	30308	Atlanta	US	03/01/202	03/01/202	Created		Courier del	3,99	2	4,99	2	6,99	27,95	
3	2	Wendy	Swanson	wendy@en	222-555-3	4957 Turke	30308	Atlanta	US	03/01/202	03/01/202	Created		Courier del	3,99	0	0	3	6,99	24,96	
4	1	James	Archer	james@em	222-555-0	1601 Laym	30308	Atlanta	US	03/01/202	03/01/202	Created		Courier del	3,99	5	4,99	0	0	28,94	

Once your e-commerce site gets some traction, this might come in handy and help you get a better picture of your business.

6.1.2 Changing order status

When a new order is created, it is by default set to status '*Created*' . When we process this order, the status will have to change accordingly. At this stage, we are able to change the order status by going into the specific order detail page and selecting a new status from the dropdown menu:



However, if we have multiple orders that need a status update, it might get cumbersome to go inside each order to switch it manually. It would also be nice if a customer could be notified when order status is set to '*Shipped*' or '*Ready for pickup*' . We don't have this kind of functionality at our disposal yet, but we already have all pieces of the puzzle that we can put together. We will have to create a new function inside the *admin.py* file to handle status change and a new task inside *tasks.py* to send out email notifications.

Let's start with changing the status of our orders. Open *admin.py* file of our *orders* application and add the following code before *OrderAdmin* class:

finesauces/orders/admin.py

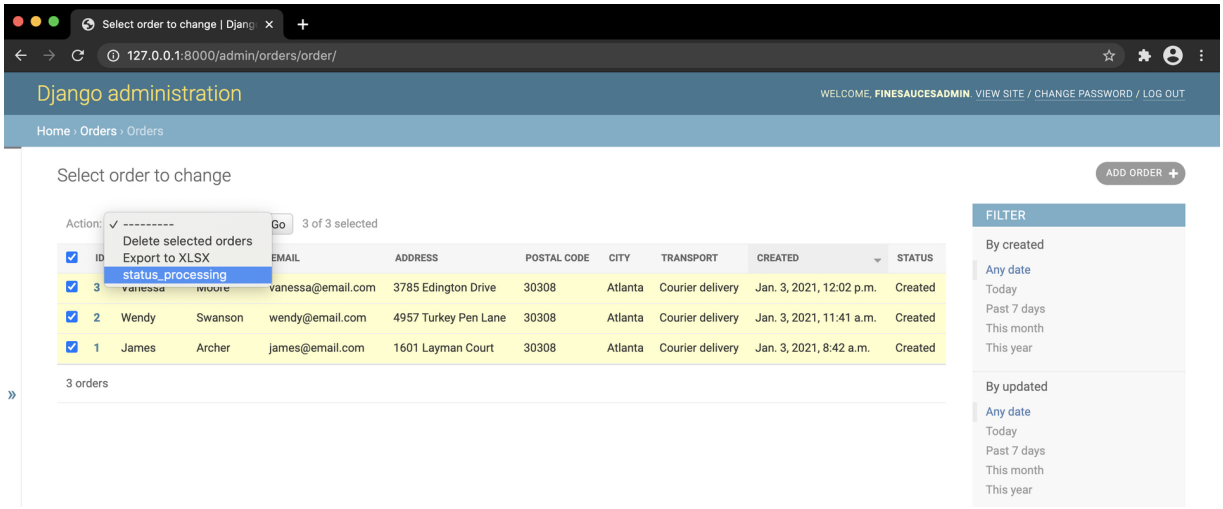
```
#...
def status_processing(modeladmin, request, queryset):
    for order in queryset:
        order.status = 'Processing'
        order.save()

status_processing.short_description = 'status_processing'
#...
```

We iterate through the provided *QuerySet* and update the status for each order. Don't forget to also update the *OrderAdmin* actions list:

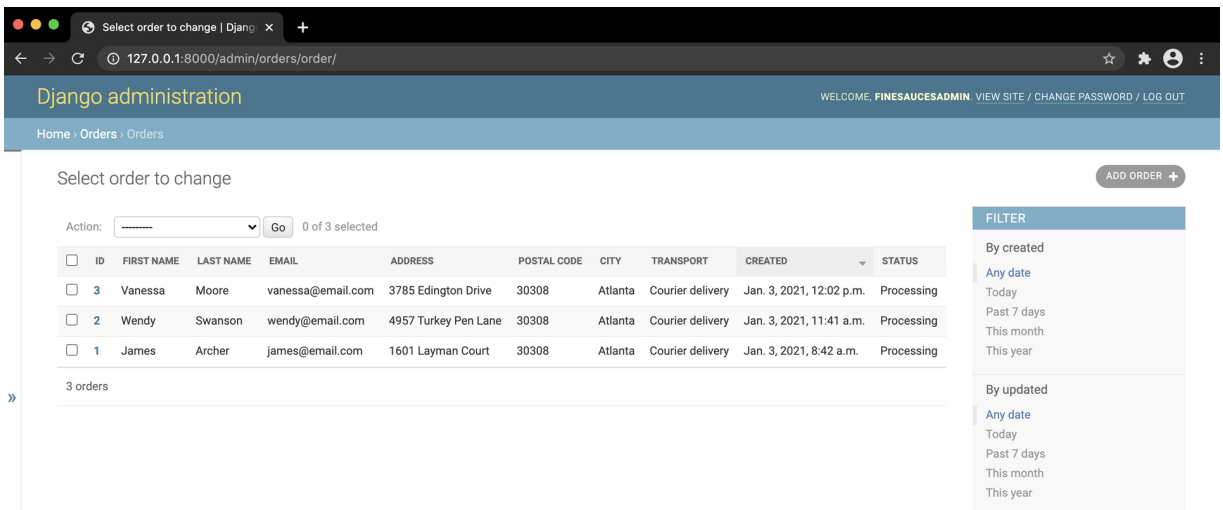
finesauces/orders/admin.py

```
#...
actions = [export_to_xlsx, status_processing ]
```

Let's test our updated administration site. Start your local development server and visit http://127.0.0.1:8000/admin/orders/order/ . Select a couple of the records, pick our new action from the dropdown, and confirm with the *Go* button.



Order status was successfully updated for all of the selected orders:



Let's implement status change for the remaining statuses: '*Shipped*', '*Ready for pickup*', and '*Ready for pickup*' .

Quick and easy, but not a very optimal way would be just to copy our *status_processing* function and modify function name and status, that is supposed to be set:

finesauces/orders/admin.py

```
#...
def status_processing(modeladmin, request, queryset):
    for order in queryset:
        order.status = 'Processing'
        order.save()

status_processing.short_description = 'status_processing'

def status_shipped(modeladmin, request, queryset):
    for order in queryset:
        order.status = 'Shipped'
        order.save()

status_shipped.short_description = 'status_shipped'
#...
```

We have a lot of code repetition going on here, and we certainly want to avoid this approach. The *for* loop inside the two functions above is almost identical, and the only difference would be the *status* that we want to be applied.

Let's go ahead and define a new function called *status_change()* :

finesauces/orders/admin.py

```
#...
def status_change(queryset, status):
    for order in queryset:
        order.status = status
```

```
        order.save()
#...
```

We basically took the for loop from our *status_processing* function and placed it inside new *status_change* function, which accepts *QuerySet* and *status* parameters. We can update our *status_processing* and *status_shipped* functions accordingly:

finesauces/orders/admin.py

```
#...
def status_change(queryset, status):
    for order in queryset:
        order.status = status
        order.save()

def status_processing(modeladmin, request, queryset):
    status_change(queryset, 'Processing')

status_processing.short_description = 'status_processing'

def status_shipped(modeladmin, request, queryset):
    status_change(queryset, 'Shipped')

status_shipped.short_description = 'status_shipped'
#...
```

We will add similar functions for remaining order statuses as well:

finesauces/orders/admin.py

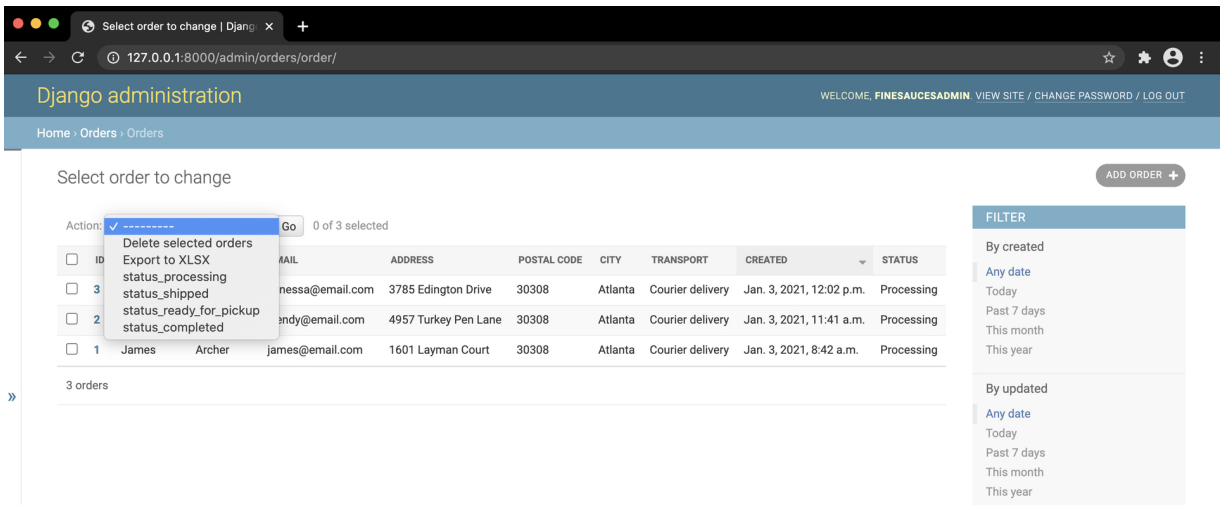
```
#...
def status_ready_for_pickup(modeladmin, request, queryset):
     status_change(queryset, 'Ready for pickup')

status_ready_for_pickup.short_description = 'status_ready_for_pickup'

def status_completed(modeladmin, request, queryset):
    status_change(queryset, 'Completed')

status_completed.short_description = 'status_completed'
#...
```

Action list in *OrderAdmin* class has to be updated accordingly:

finesauces/orders/admin.py

```
#...
actions = [export_to_xlsx,
           status_processing,
           status_shipped,
           status_ready_for_pickup,
           status_completed]
```

Let's refresh the admin site to reflect our latest update:



Now we need to let the user know when an order status change occurs. We will create a new task inside our *tasks.py* file residing in the *orders* directory. Go ahead and add the following code to the file:

finesauces/orders/tasks.py

```
#...
@task
def status_change_notification(order_id):
    order = Order.objects.get(id=order_id)
    subject = f'Order nr. {order.id}'
    message = f'Dear {order.first_name},\n\n'\
              f'Status of your order {order.id} was changed to {order.status}'
```

```
    mail_sent = send_mail(
        subject,
        message,
        'eshop@finesauces.store',
        [order.email]
    )

    return mail_sent
```

This task is very similar to the *order_created* task. We could probably do some refactoring here, but the *order_created* task will be adjusted in the upcoming section.

Return to *admin.py* file. Import *status_change_notification* task we just created and add it to *status_change* function as follows:

finesauces/orders/admin.py
```
from .tasks import status_change_notification
#...
def status_change(queryset, status):
    for order in queryset:
        order.status = status
        order.save()

        status_change_notification.delay(order.id)
        #...
```

Open another terminal and start Celery worker from your *finesauces_project* directory using the following command if it's not running already:

terminal
```
(env) ~/finesauces$ celery -A finesauces_project worker -l info
```

When you change order status now, the customer will be notified about this update.

6.2 Generating PDF invoices

Now that customers can create orders and receive email notifications when they do so, it would be great to create some invoices that we would have available in admin site and which users will receive attached to the email.

We will first create an HTML template, render it using Django and pass it to WeasyPrint to generate our PDF file. *WeasyPrint* [48] is a visual rendering engine for HTML and CSS that can export to PDF and PNG.

6.2.1 Installing WeasyPrint

For WeasyPrint to work properly, we need to install the required dependencies first. Visit the https://weasyprint.readthedocs.io/en/stable/index.html for installation instructions for your platform. For Debian or Ubuntu based Linux, use:

terminal
```
(env) ~/finesauces$ sudo apt-get install
build-essential python3-dev python3-pip python3-setuptools
python3-wheel python3-cffi libcairo2 libpango-1.0-0
libpangocairo-1.0-0 libgdk-pixbuf2.0-0 libffi-dev
shared-mime-info
```

If you are running macOS, you can use the Homebrew manager:

terminal
```
(env) ~/finesauces$ brew install cairo pango gdk-pixbuf libffi
```

WeasyPrint itself is available for download as a Python library. Therefore we can go ahead and install it using the *pip* package manager:

terminal
```
(env) ~/finesauces$ pip install WeasyPrint
```

6.2.2 Creating PDF invoice template

Let's create an HTML template that will be used to render our PDF invoice. Create a new file inside the *templates* directory within the *orders* application and name it *pdf.html* . Add the following code to it:

finesauces/orders/templates/pdf.html
```
<html>

<body>
  <div>
    <span class="h3 font-weight-bold muted">
      finesauces.store
    </span>
    <span class="float-right font-weight-bold">
      Order #{{ order.id }}
    </span>
  </div>
```

```
  <div class="mt-3">
    Created: {{ order.created|date }}
  </div>

  <hr>

  <div id="container">
    <div class="item mt-2">
      <div class="font-weight-bold">
        Manufacturer
      </div>

      E-mail: eshop@finesauces.store<br>
      Internet: finesauces.store<br>
      Telephone: 404-555-0210<br>
    </div>

     <div class="item mt-2">
      <div class="font-weight-bold">
        Customer
      </div>
      {{ order.first_name }} {{ order.last_name }}<br>
      {{ order.address }}<br>
      {{ order.postal_code }}, {{ order.city }}
    </div>
  </div>

  <hr>

  <h3>Your order:</h3>
  <table class="table mt-3">
    <thead>
      <tr>
        <th>Product</th>
        <th class="text-right">Price</th>
        <th class="text-right">Quantity</th>
        <th class="text-right">Price total</th>
      </tr>
    </thead>
    <tbody>
    {% for item in order.items.all %}
      <tr>
        <td>{{ item.product.name }}</td>
        <td class="num text-right">${{ item.price }}</td>
        <td class="num text-right">{{ item.quantity }}x</td>
        <td class="num text-right">${{ item.get_cost }}</td>
      </tr>
    {% endfor %}
      <tr>
        <td colspan=3>Transport - {{ order.transport }}</td>
        <td class="num text-right">${{ order.transport_cost }}</td>
      </tr>
      <tr class="total font-weight-bold">
        <td colspan="3">Price total:</td>
        <td class="num text-right">
          ${{ order.get_total_cost|floatformat:2 }}
        </td>
      </tr>
    </tbody>
   </table>
</body>

</html>
```

6.2.3 Rendering PDF invoices

Our HTML template is ready to be rendered. We are now going to work on view to generate our PDF invoices for existing orders. Edit the *views.py* file inside the *orders* application and add these imports and code for *invoice_pdf* :

finesauces/orders/views.py

```
#...
from django.contrib.admin.views.decorators import staff_member_required
from django.http import HttpResponse
from django.template.loader import render_to_string
import weasyprint
#...
@staff_member_required
def invoice_pdf(request, order_id):
    order = get_object_or_404(Order, id=order_id)

    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = f'filename=order_{order.id}.pdf'

    # generate pdf
    html = render_to_string('pdf.html', {'order': order})
    stylesheets=[weasyprint.CSS(settings.STATIC_ROOT + 'css/pdf.css')]
    weasyprint.HTML(string=html).write_pdf(response,stylesheets=stylesheets)
```

```
    return response
    #...
```

This is our view for generating PDF invoices. We utilize *@staff_member_required decorator* to limit access only to admin/staff members. We then retrieve our *Order* object based on the *order_id* provided and generate a new *HttpResponse* object which specifies the *application/pdf* content type and includes the *Content-Disposition* header to set the filename. Then the template is rendered and stored in the *html* variable. We use the static file *css/pdf.css* to add CSS styles for PDF file. *WeasyPrint* generates a PDF file from the rendered HTML code and writes the file to the *HttpResponse* object.

Since we need to use the *STATIC_ROOT* setting, we have to add it to our project. This is the project's path where static files reside. Edit the *settings.py file* inside *finesauces_project* and add the following setting:

finesauces/finesauces_project/settings.py

```
#...
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
#...
```

Then run the following command:

terminal

```
(env) ~/finesauces$ python manage.py collectstatic
```

You should see a similar message in your terminal:

terminal

```
135 static files copied to '/Users/peter/finesauces/static'.
```

The *collectstatic* command collects static files from each of your applications (and any other places you specify) into a single location that can easily be served in a production environment. All static files from our applications are copied into the directory defined in the *STATIC_ROOT* setting *(STATIC_ROOT = os.path.join(BASE_DIR, 'static/'))*.

Now we need to map our *invoice_pdf* view to the URL pattern. Edit the *urls.py* file in the *orders* application and add the following URL pattern to it:

finesauces/orders/urls.py

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

app_name = 'orders'

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

To display the invoice URL for each order record, edit the *admin.py* file inside the *orders* application and add the following code above the *OrderAdmin* class:

finesauces/orders/admin.py

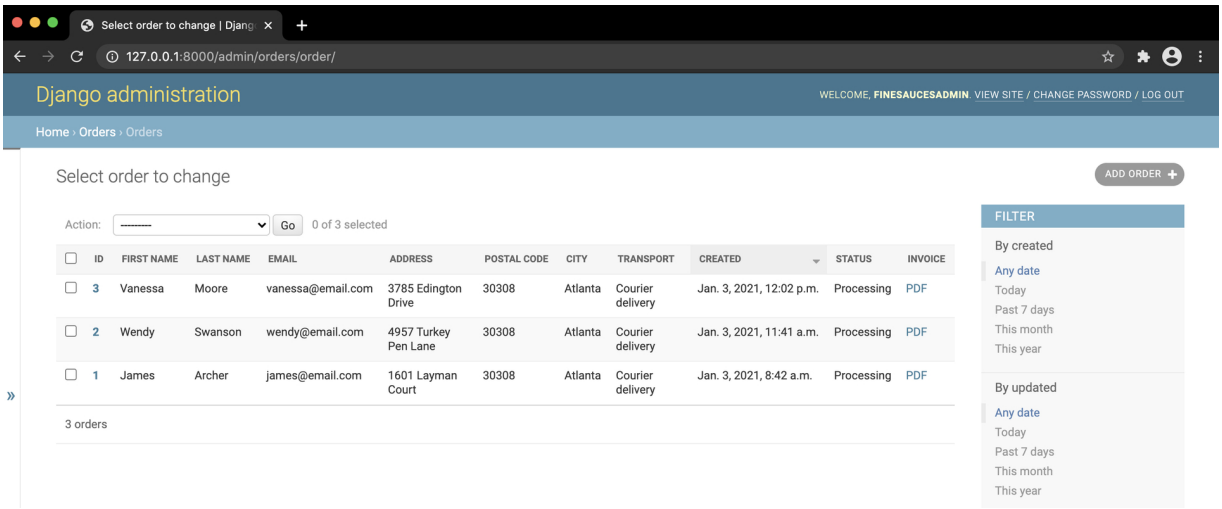
```
#...
from django.urls import reverse
from django.utils.html import format_html

def order_pdf(obj):
    return format_html('PDF',
        reverse('orders:invoice_pdf', args=[obj.id]))

order_pdf.short_description = 'Invoice'
```

We use *format_html* function to mark our HTML as safe to be rendered in the template. Django by default does not trust any HTML code and will escape it before placing it in the output. This behavior prevents Django from outputting potentially dangerous HTML and allows us to create exceptions for returning safe HTML. By using *short_description*, we specify the column name to be set in the admin area. If we omitted this setting, column name would be set to *ORDER PDF* by default. As a last step, we need to add *order_pdf* to the *list_display* attribute of the *OrderAdmin* class:

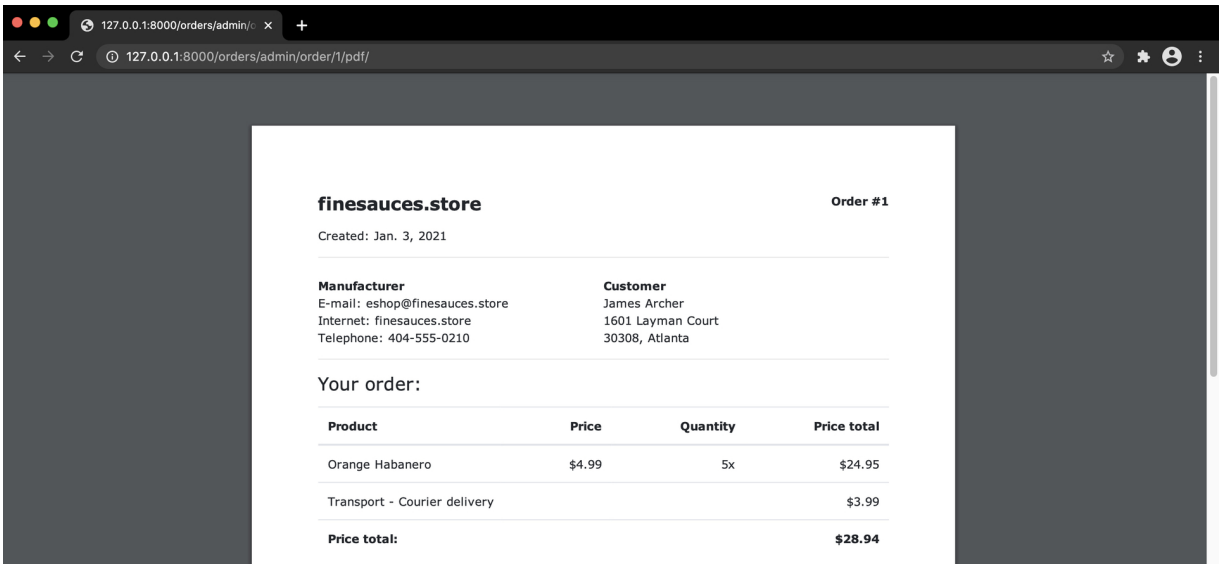
finesauces/orders/admin.py

```
#...
@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = [
        'id', 'first_name', 'last_name', 'email',
        'address', 'postal_code', 'city',
        'transport', 'created', 'status', order_pdf
    ]
    #...
```

If you visit http://127.0.0.1:8000/admin/orders/order/ , you will notice new column *INVOICE* was added for each Order record:



Feel free to click on any of the *PDF* links to view the invoice:



Styling for invoices template is defined in the *listings/static/css/pdf.css* file.

6.2.4 Sending invoices by email

When a new order is created, a customer already receives an email notification. This notification, however, lacks the invoice, which might be useful to the customer. Let's go ahead and include this PDF invoice as an email attachment.

Open *tasks.py* file of the *orders* application and add the following imports:

finesauces/orders/tasks.py

```
from django.core.mail import EmailMessage
from django.template.loader import render_to_string
from django.conf import settings
from io import BytesIO
import weasyprint
```

Then modify our asynchronous *order_created* task to look like this:

finesauces/orders/tasks.py

```
@task
def order_created(order_id):
    order = Order.objects.get(id=order_id)
    subject = f'Order nr. {order.id}'
    message = f'Dear {order.first_name}, \n\n' \
              f'Your order was successfully created.\n' \
              f'Your order ID is {order.id}.'

    email = EmailMessage(
        subject,
        message,
        'eshop@finesauces.store',
        [order.email]
    )

    # generate pdf
    html = render_to_string('pdf.html', {'order': order})
    out = BytesIO()
    stylesheets = [weasyprint.CSS(settings.STATIC_ROOT + 'css/pdf.css')]
     weasyprint.HTML(string=html).write_pdf(out,stylesheets=stylesheets)
```

```
    # attach PDF file
    email.attach(f'order_{order.id}.pdf',
        out.getvalue(),
        'application/pdf'
    )

    # send e-mail
    email.send()
```

In this task, we use the *EmailMessage* class provided by Django to create an email object. A generated PDF file is rendered and placed into *BytesIO* instance, which is an in-memory bytes buffer. Then, we attach the PDF file to the *EmailMessage* object using the *attach()* method, including the contents of the *out* buffer. Finally, we send the email.

6.3 Admin site styling

Django admin sites, by default, look the same for all Django projects. They all share the same styling and same navbar with the *Django administration* logo. There's nothing wrong with that, but it would be nice if we could spice things up a little and make our Admin site a little bit more exciting and match our e-commerce site theme. Lucky for us, it's pretty simple to modify the styling of the Admin site.

We need to create a *templates* folder inside a base directory, directly within the *finesauces* folder, next to our project folder *finesauces_project, listings, cart,* and *orders* applications. Within the *templates* folder, create an *admin* subfolder and within it, a file called *base_site.html* . You should end up with the following folder structure:

```
finesauces
└── templates
    └── admin
        └── base_site.html
```

Add the following code to the *base_site.html* file:

finesauces/templates/admin/base_site.html

```
{% extends 'admin/base.html' %}
{% load static %}

{% block branding %}

  <h1 id="head">
    finesauces
  </h1>

{% endblock %}
{% block extrastyle %}

  <link rel="stylesheet"
    href="https://fonts.googleapis.com/css2?family=Nunito&display=swap">
  <link rel="stylesheet" href="{% static 'css/admin.css' %}">

{% endblock %}
```

Now we need to let Django know about our new template. Since it is not located in a standard templates directory within any of our apps, we need to modify the *settings.py* file accordingly. Modify *DIRS* attribute to look like this:

finesauces/finesauces_project/settings.py

```
#...
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(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',
                'finesauces_project.context_processors.cart',
            ],
         },
    },
]
#...
```

Let's also create styling sheet for our admin site. Create *admin.css* file inside *listings/static/css* directory like such:

```
listings
└── static
    ├── css
    │   ├── admin.css
    │   ├── pdf.css
    │   └── style.css
    └── js
```

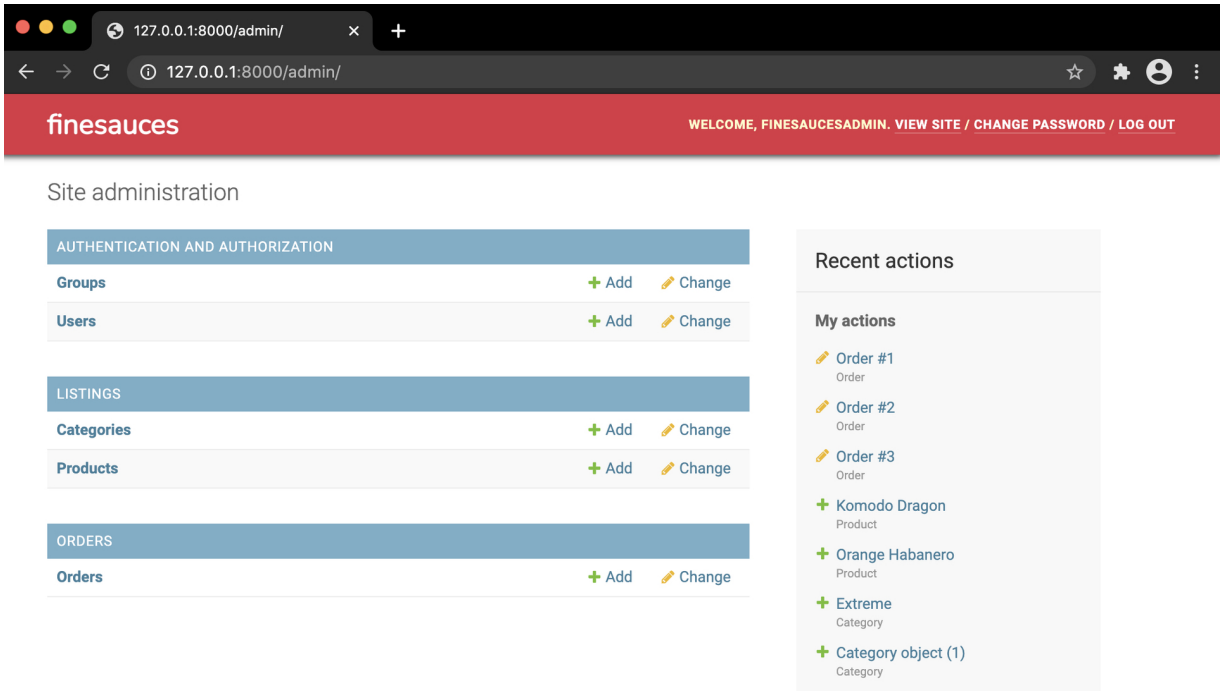
```
        └── scripts.js
```

and add in the following CSS styling:

finesauces/listings/static/css/admin.css

```
:root {
    --primary-white: #fff;
    --primary-orange: #ffc107;
    --primary-red: #dc3545;
    --primary-grey: #444;
}

#header {
    background: var(--primary-red);
    box-shadow: 0px 2px 5px #444;
    z-index: 2;
}

#header #branding #head {
    font-weight: 700 !important;
    font-family: 'Nunito', sans-serif;
    color: var(--primary-white) !important;
}

#user-tools {
     font-weight: bold;
}
```

Let's check our progress so far. Make sure your development server is running and open your admin site at http://127.0.0.1:8000/admin/ :

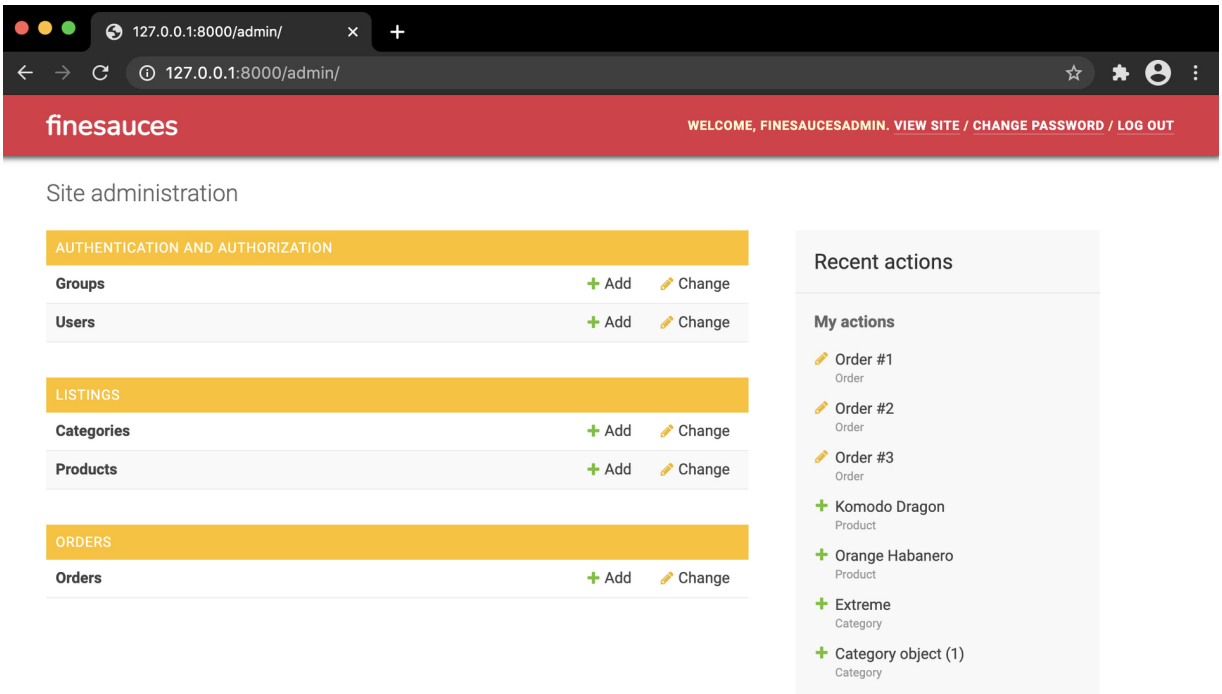


Our navbar matches the theme of our application! Let's change our apps section's background color from blue to orange and font color to dark grey. Add the following styling to the *admin.css* file:

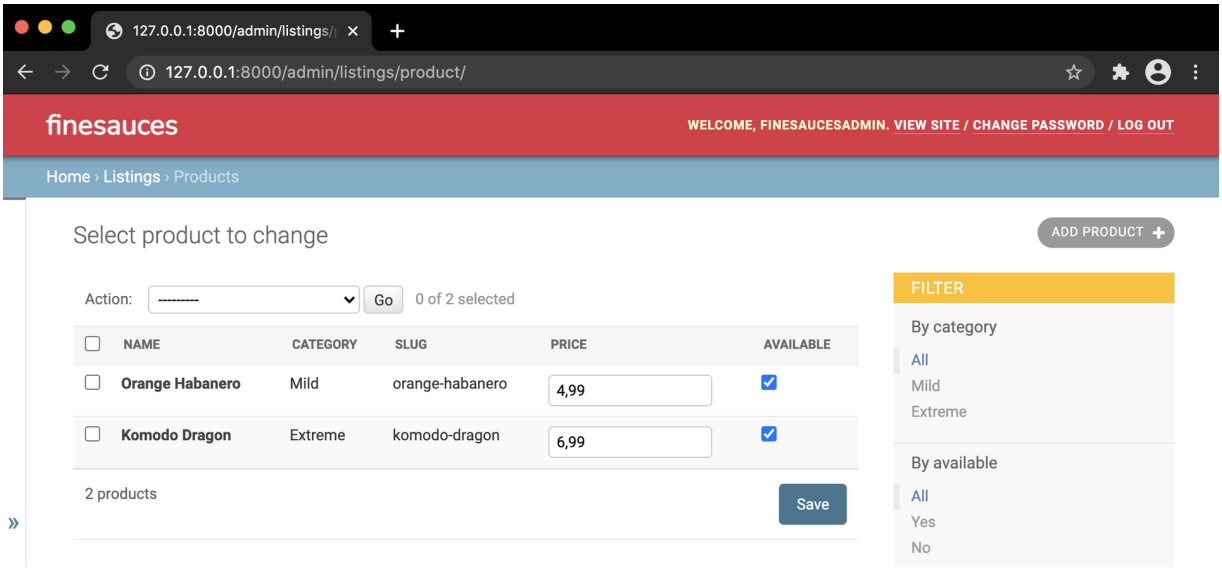
finesauces/listings/static/css/admin.css

```
#...
.module h2, .module caption, .inline-group h2 {
    background: var(--primary-orange);
}

.module a {
    color: var(--primary-grey);
}
```

The updated version of admin site so far:



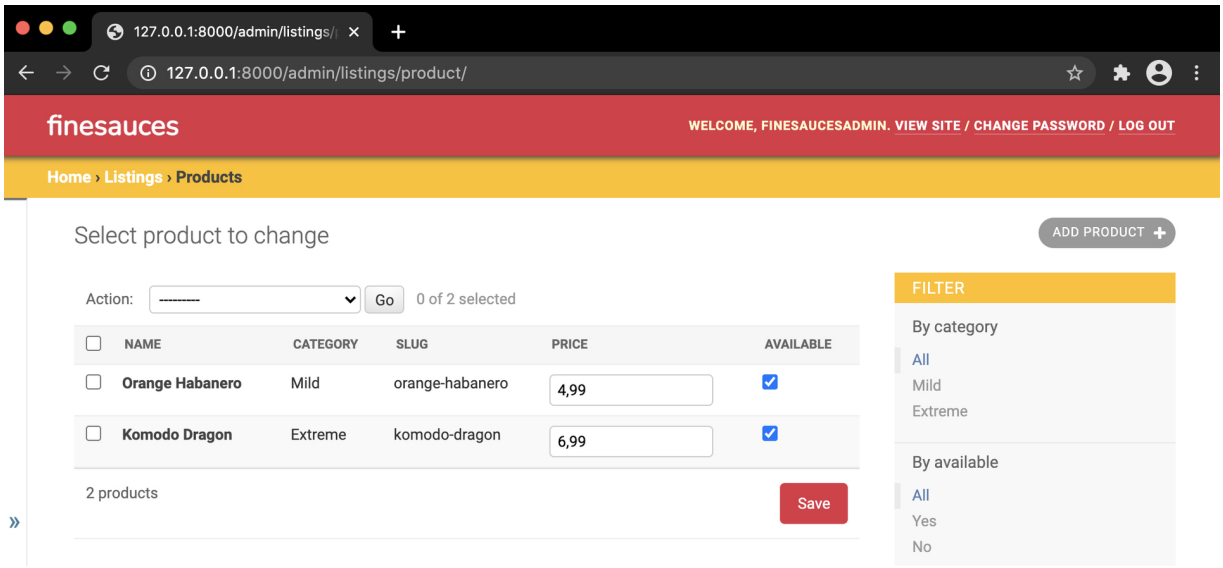
Let's work on our *models* sections now. At this stage, they share the following styling:



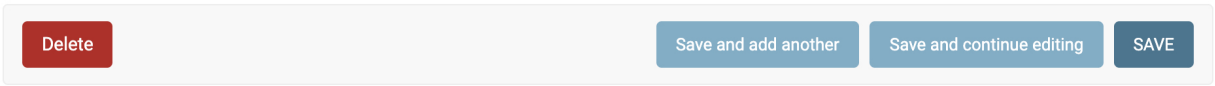
I would like to update this section to match our theme as well. Add the following styling to the *admin.css* file to modify breadcrumbs colors:

```
finesauces/listings/static/css/admin.css
#...
div.breadcrumbs {
    background: var(--primary-orange);
     color:var(--primary-grey);
    font-weight: bold;
}

.paginator input[type=submit]{
    background-color: var(--primary-red);
}
```

Looks much better now:



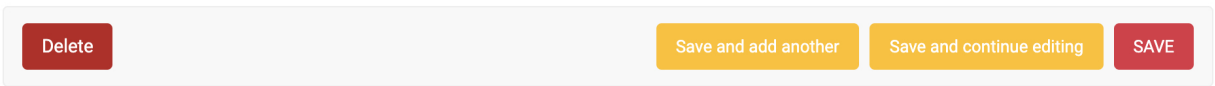
The last element I want to style are buttons at the bottom of the models sections:



Add the following styling to our *admin.css* file:

```
finesauces/listings/static/css/admin.css
#...
.submit-row input.default {
    background-color: var(--primary-red);
}

.submit-row input:hover.default {
    background-color: #c23442;
}

.submit-row input, a.button {
    background-color: var(--primary-orange);
}

.submit-row input:hover, a.button:hover {
    background-color: #ffac07;
};
```

Save buttons now match the orange color applied throughout our project.



In this optional chapter, we styled the admin site to match our e-commerce site theme. It's a great way to breathe a little something extra into your project and make it really stand out! There is, of course, a lot more you can customize. Feel free to go around the admin site, inspect the elements you would like to style, and add your styling to the *admin.css* file.

7 Customer accounts

So far, we have created a very solid e-commerce project. Users can browse through our products, leave a review, create an order, check out the cart, and pay by card. The last piece we are missing is customer accounts. To provide a more personal experience, it would be great if users could view their previous purchases, invoices, and store their account information. This would also enable us to prefill the checkout form with available user data based on the account information.

Let's start by making it possible for users to login, logout, register, change their password, and reset it. Luckily, Django comes with an already built-in user authentication [49] framework bundled in *django.contrib.auth* module. It is by default included in our settings file generated when we start our project.

Let's create an *accounts* application to manage our user authentication:

terminal
```
(env) ~/finesauces$ python manage.py startapp accounts
```

Now we need to activate this application by adding it to the *INSTALLED_APPS* list in the *settings.py file* in the *finesauces_project* directory. Make sure you place it as first, before any other app:

finesauces/finesauces_project/settings.py
```
#...
INSTALLED_APPS = [
    'accounts.apps.AccountsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
     'django.contrib.staticfiles',
    'listings.apps.ListingsConfig',
    'cart.apps.CartConfig',
    'orders.apps.OrdersConfig',
]
#...
```

By placing our application first in the *INSTALLED_APPS,* we ensure that our authentication templates will be used by default instead of any other authentication templates contained in other applications. Django looks for templates by order of the application appearance in the *INSTALLED_APPS* setting.

7.1 Login view

We first want to enable users to log in to our site. This view will need to retrieve the user email and password submitted in the login form, authenticate the user, log the user in, and start the authenticated session.

Let's work on the login form first. Create *forms.py* file inside the *accounts* application directory and add the following code to it:

finesauces/accounts/forms.py
```
from django import forms

class LoginForm(forms.Form):
    email = forms.CharField(
        widget=forms.EmailInput(attrs={
            'class': 'form-control'
        })
    )
    password = forms.CharField(
        widget=forms.PasswordInput(attrs={
            'class': 'form-control'
        })
    )
```

We specify that for the email attribute, we want Django to generate an HTML element with email type and password type for the password field.

As for the login view, open the *views.py* file and add the following content inside:

finesauces/accounts/views.py
```
from django.http import HttpResponse
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login
from django.contrib.auth.models import User
from .forms import LoginForm

def user_login(request):
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            cf = form.cleaned_data
            user_record = None
            try:
                user_record = User.objects.get(email=cf['email'])
            except User.DoesNotExist:
```

```
                pass

            if user_record:
                user = authenticate(
                    request,
                    username=user_record,
                    password=cf['password']
                )

                if user is not None:
                    login(request, user)
                    return redirect('listings:product_list')

    else:
        form = LoginForm()
    return render(request, 'accounts/login.html', {'form': form})
```

If we receive the *GET* request, that is, when a user visits our login page without submitting anything, we simply render the login form. When a user submits data using the *POST* method, we instantiate, validate, and clean the form. We then look for the user record based on the email provided. If no such user is found, we raise the *Model.DoesNotExist* [50] exception and render *login.html* template again. We could also use the shortcut function *get_object_or_404()* to handle this, but we don't want to throw a 404 error at the user.

If a record with the provided email exists, we try to verify user credentials using *authenticate()* [51] provided by the Django authentication framework. Once a user is successfully verified, *authenticate()* returns *User* object, we attach it to the current session by using *login()* [52] function and redirect to our landing page with products.

Now we need to create a URL pattern for this view. Create a *urls.py* file inside the *accounts* application directory and add the following code to it:

finesauces/accounts/urls.py

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

urlpatterns = [
    path('login/', views.user_login, name='login'),
]
```

Edit the main *urls.p* y file located in the *finesauces_project* directory and add the URL patterns of the *account* application to the top:

finesauces/finesauces_project/urls.py

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

As a final step, let's go ahead and create an HTML template to display our login form. Inside *accounts* application, create *templates* folder and within it *accounts* folder, which will contain our *login.html* file.

```
templates
└── accounts
    └── login.html
```

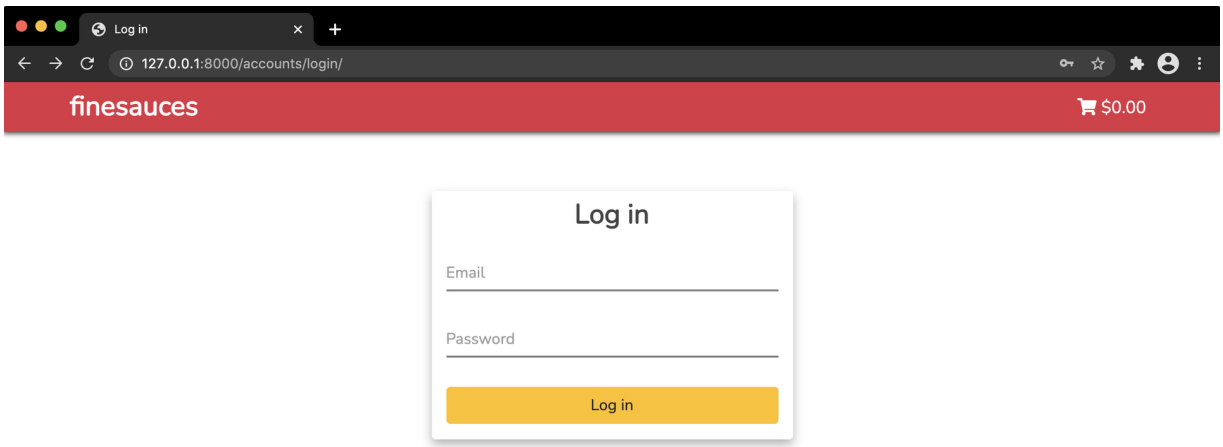
Open the *login.html* file and add make it look like:

finesauces/accounts/templates/accounts/login.html

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

{% block title %}Log in{% endblock %}

{% block content %}
<div class="card shadow-custom border-0 col-lg-4 mx-auto mb-3">
  <h3 class="py-2 font-weight-bold text-grey text-center">Log in</h3>
  <form action="" method="post">
    {% csrf_token %}
    <div class="input-field">
      {{ form.email }}
      <label for="email">Email</label>
    </div>
    <div class="input-field">
      {{ form.password }}
      <label for="password">Password</label>
    </div>
    <button class="btn btn-block btn-warning mb-3" type="submit">
      Log in
    </button>
  </form>
```

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

Make sure your development server is running and check our login page at http://127.0.0.1:8000/accounts/login/ .



We already created our admin superuser. If you did set an email address (*in case not, visit* http://127.0.0.1:8000/admin/auth/user/ *section, find your user and add any email address)*, go ahead, fill in the form and click on *Log in*.

If everything works out, we will be redirected to our landing product page. The last important update we need to do here is in the *settings.py* file in the *finesauces_project* directory. Open it and add the following code to the end of the file in front of the *local_settings* import:

finesauces/finesauces_project/settings.py

```
#...
LOGIN_URL = 'login'
#...
```

This setup is critical for views that require a user to be logged in to access them. *LOGIN_URL* specifies where the user should be redirected to log in.

7.1.1 Messages framework

When users interact with our platform, there are cases where we might want to inform them about the outcome of their actions. Django comes with a built-in *messages* [53] framework that allows us to display notifications to the users.

It is located in *django.contrib.messages* and is also included by default in the list of the *settings.py* file when we create new projects.

The messages framework provides a simple way to display messages to users. It allows us to temporarily store messages in request and retrieve them for display in a subsequent one. We can use the *messages* framework in our views by importing the messages module and adding new messages with simple shortcuts.

Let's create a message to inform users when they provide invalid credentials. Open the *views.py* file in the *accounts* directory and import the *messages* module:

finesauces/accounts/views.py

```
from django.contrib import messages
#...
```

Add the message in the following manner:

finesauces/accounts/views.py

```
#...
if user_record:
    user = authenticate(
        request,
        username=user_record,
        password=cd['password']
    )

    if user is not None:
        login(request, user)
        return redirect('listings:product_list')

messages.error(request, 'Incorrect email / password')
#...
```

Every message can be tagged with a specific tag that determines its priority and purpose. We are adding an *error* tag for this case. There are five tags in total: *debug* , *info* , *success* , *warning*, and *error* . We can use those to differentiate between messages passed on to the template. Then, within the template, we can check specific tags with the following notation:

```
{% if message.tags == 'error' %}
```

To utilize the error message that we just implemented, open *login.html* template and add the following code right to the top, after opening *{% block content %}* tag:

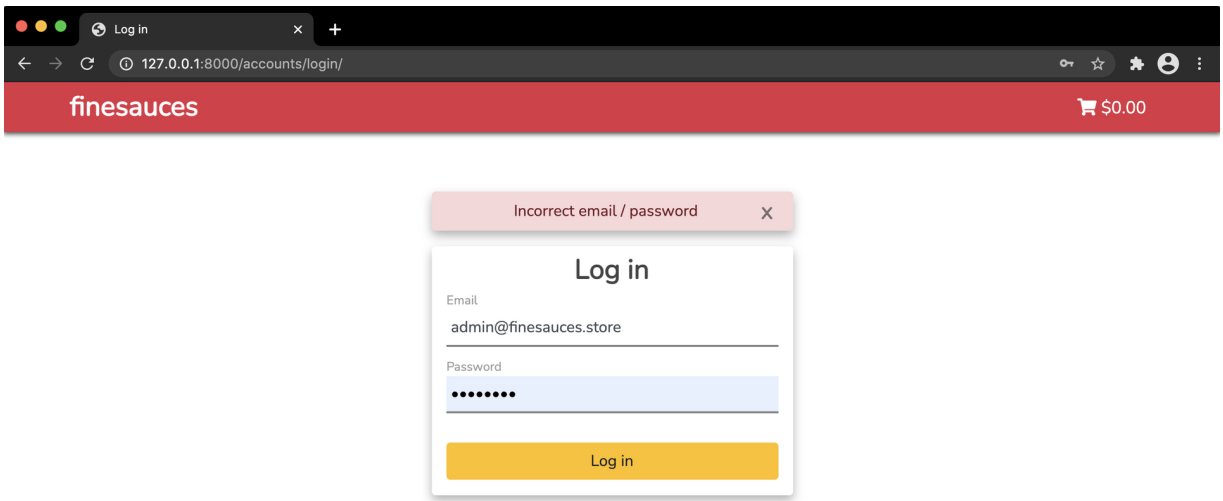
```
finesauces/accounts/templates/accounts/login.html
#...
{% block content %}
{% if messages %}
  <div class="shadow-custom messages alert alert-danger
    col-lg-4 mx-auto text-center">
    {% for message in messages %}
      {{ message }}
      <a href="" class="close">x</a>
    {% endfor %}
  </div>
{% endif %}
#...
```

If we were looking for an *error* tag specifically, the loop above could be modified as follows:

```
finesauces/accounts/templates/accounts/login.html
#...
{% block content %}
{% if messages %}
<div class="shadow-custom messages alert alert-danger col-lg-4 mx-auto text-center">
  {% for message in messages %}
    {% if message.tags == 'error' %}
      {{ message }}
      <a href="" class="close">x</a>
    {% endif %}
  {% endfor %}
</div>
{% endif %}
#...
```

We don't have to perform this check in the *login.html* template. We will, however, utilize this in a later section.

Let's test our updated login form. Open http://127.0.0.1:8000/accounts/login/ and fill in incorrect data. Once you try to log in, you will receive a following error message:



7.1.2 Django LoginView

In the previous section, we created our custom login view. However, as we stated at the beginning of this chapter, Django comes with a built-in user authentication framework that we can utilize. To take advantage of that, we need to update our *urls.py* file inside the *accounts* application to look like this:

```
finesauces/accounts/urls.py
from django.urls import path
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/', auth_views.LoginView.as_view(), name='login'),
]
```

We replaced our custom login view with *LoginView* [54] . Now we need to create a *registration* folder inside the *templates* directory and create a *login.html* template inside. This is the default location where *LoginView* will search for the *login.html* template.

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

Go ahead and fill in the following code:

finesauces/accounts/templates/registration/login.html

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

{% block title %}Log in{% endblock %}

{% block content %}
{% if form.errors %}
  <div class="shadow-custom messages alert alert-danger
    col-lg-4 mx-auto text-center">
    Incorrect username / password
      <a href="" class="close">x</a>
  </div>
{% endif %}

<div class="card shadow-custom border-0 col-lg-4 mx-auto mb-3">
  <h3 class="py-2 font-weight-bold text-grey text-center">Log in</h3>
  <form action="" method="post">
    {% csrf_token %}
    <div class="input-field">
      {{ form.username }}
      <label for="username">Username</label>
    </div>
    <div class="input-field">
      {{ form.password }}
      <label for="password">Password</label>
    </div>
    <button class="btn btn-block btn-warning mb-3" type="submit">
      Log in
    </button>
  </form>
</div>
{% endblock %}
```

Visit http://127.0.0.1:8000/accounts/login/ and try to sign in. Don't forget to use username instead of user email in this case! Upon logging in, you will be redirected to:

http://127.0.0.1:8000/accounts/profile/ . Why? This is where *LoginView* tries to redirect us by default. Let's change that and set it to redirect us to our landing product list page. Open *settings.py* file inside *finesauces_project* directory and add the following code to the end of the file:

finesauces/finesauces_project/settings.py

```
#...
LOGIN_URL = 'login'
LOGIN_REDIRECT_URL = 'listings:product_list'
#...
```

When we try to log in now, we will be redirected to our landing page. At this stage, we might ask ourselves whether we should build our own login functionality or utilize the *LoginView* solution. As with many things, it really depends on the use case. *LoginView* represents a quick and straightforward authentication solution. It is, however, very limited in the customization area. On the other hand, custom login views open many more possibilities in this regard. They would be the way to go when some specific actions are to be performed before, during, or after the login process. For the remaining part of this book, we will be using the previous, custom solution.

7.2 Logout view

Users are now able to log in to our site. Let's enable them to log out as well. Thanks to the Django auth system, this is very simple to achieve. We only need to set the *logout* URL in the *urls.py* file.

Open urls.py file in the accounts directory and add the following code:

finesauces/accounts/urls.py

```
from django.urls import path
from . import views
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/', views.user_login, name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout')
]
```

We also need to specify where Django is supposed to redirect users after logout. Let's redirect them to the landing page. To do so, we need to update the *settings.py* file inside the *finesauces_project* directory. Add the following code to the end of the file:

finesauces/finesauces_project/settings.py

```
#...
LOGIN_URL = 'login'
LOGOUT_REDIRECT_URL = 'listings:product_list'
#...
```

With the current setup, users are able to log in and log out. In theory, at least. However, when they visit our site, it will not be possible for them to locate our authentication form as we did not provide any links leading to it.

Let's update the *base.html* template inside the *listings* application to display *login* and *cart* links when users are logged out. When they log in, we will show their *username*, *logout* link, and *cart* as well.

Find the following section in the *base.html* file:

finesauces/listings/templates/base.html

```
#...
<div class="collapse navbar-collapse" id="navbarCollapse">
  <ul class="navbar-nav ml-auto">
    <li class="nav-item">
      <div class="dropdown-custom">
         <a href="{% url 'cart:cart_detail' %}"
        #...
```

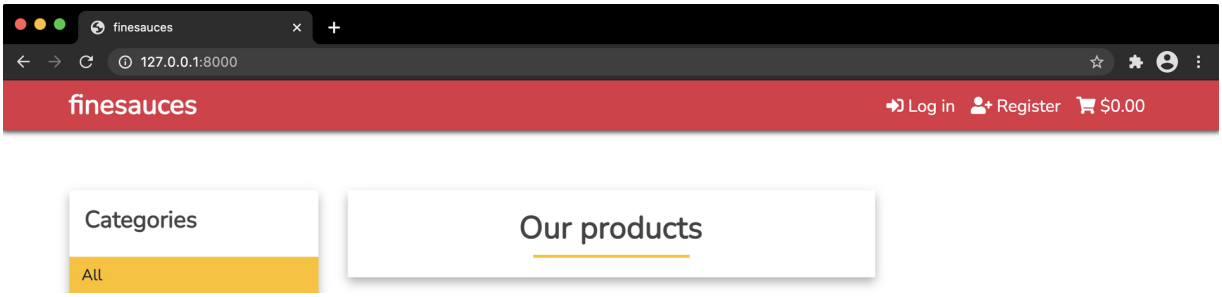
Insert this code between <*ul*> and <*li*> tags as follows:

finesauces/listings/templates/base.html

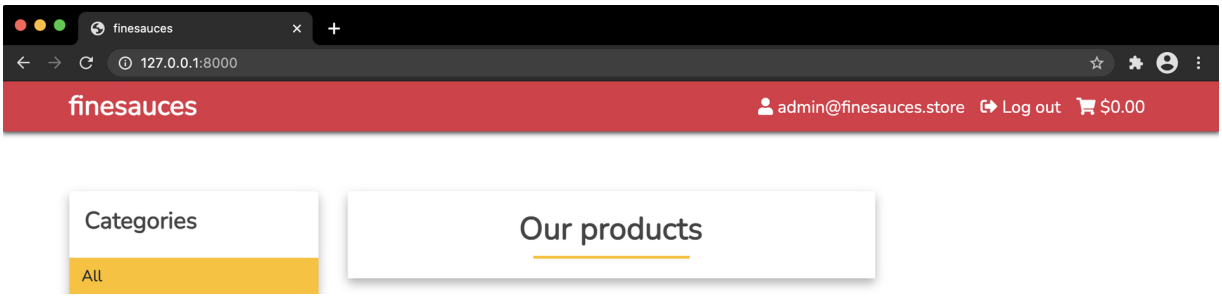
```
#...
<div class="collapse navbar-collapse" id="navbarCollapse">
  <ul class="navbar-nav ml-auto">
    {% if request.user.is_authenticated %}
      <li class="nav-item">
        <a href="" class="nav-link text-white">
          <i class="fas fa-user"></i> {{ request.user.email }}
        </a>
      </li>
      <li class="nav-item">
        <a href="{% url 'logout' %}" class="nav-link text-white">
          <i class="fas fa-sign-out-alt"></i> Log out
        </a>
      </li>
      {% else %}
      <li class="nav-item">
        <a href="{% url 'login' %}" class="nav-link text-white">
          <i class="fas fa-sign-in-alt"></i> Log in
        </a>
      </li>
      <li class="nav-item">
        <a href="" class="nav-link text-white">
          <i class="fas fa-user-plus"></i> Register
        </a>
      </li>
    {% endif %}
    <li class="nav-item">
      <div class="dropdown-custom">
        <a href="{% url 'cart:cart_detail' %}"
        #...
```

To display appropriate links, we have to determine whether the user is logged in or not. The current user is set in the *HttpRequest* object by the authentication middleware. We can access it with a *request.user* notation. *User* object is present in the request even if a user is not authenticated. Not authenticated user is set in the request as an instance of *AnonymousUser* [55] . The best way to check whether the user is authenticated is by accessing the read-only attribute *is_authenticated* .

Let's review our updated site. When a user is logged out, the navbar will look like this:



When a user successfully logs in, navbar links will change accordingly:



7.3 User registration

Existing users can now log in and log out. Now we will build a view to allow visitors to create their accounts. We will start with the registration form. Open *forms.py* file located inside the *account* application directory and add the following code to it:

```
finesauces/accounts/forms.py
#...
from django.contrib.auth.models import User

class UserRegistrationForm(forms.ModelForm):

    password2 = forms.CharField(
         widget=forms.PasswordInput(attrs={
            'class': 'form-control',
        }))

    class Meta:
        model = User
        fields = ('first_name', 'last_name', 'email', 'password')

        widgets = {
            'first_name': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'last_name': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'email': forms.EmailInput(
                attrs={'class': 'form-control'}
            ),
            'password': forms.PasswordInput(
                attrs={'class': 'form-control'}
            ),
        }
        #...
```

Import our *UserRegistrationForm* within *views.py* file and create *register* view:

```
finesauces/accounts/views.py
#...
from .forms import LoginForm, UserRegistrationForm

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            cf = user_form.cleaned_data

            email = cf['email']
            password = cf['password']
            password2 = cf['password2']

             if password == password2:
                if User.objects.filter(email=email).exists():
                    messages.error(request,
                        'User with given email already exists')
                    return render(
                        request,
                        'accounts/register.html',
                        {'user_form': user_form}
                    )
            else:
                messages.error(request, 'Passwords don\'t match')
                return render(
                        request,
                        'accounts/register.html',
                        {'user_form': user_form}
                    )
            # Create a new user object
            new_user = User.objects.create_user(
                first_name=cf['first_name'],
                last_name=cf['last_name'],
                username=email,
                email=email,
                password=password
```

```
            )

            return render(
                request,
                'accounts/register_done.html',
                {'new_user': new_user}
            )
    else:
        user_form = UserRegistrationForm()
    return render(
        request,
        'accounts/register.html',
        {'user_form': user_form}
    )
```

When the *UserRegistrationForm* form is submitted via *POST* action, we instantiate, validate, and clean it. During the registration process, a user submits the password twice. If the provided passwords don't match, we send back an error message informing the user about this issue. If the passwords match, we check if a user with the provided email already exists. If a user with this email already exists, we let the user know via an error message. In case no issues arise, we proceed to the user creation and render the *register_done.html* template.

Now let's add a new URL pattern to our *urls.py* file inside the *accounts* application:

finesauces/accounts/urls.py

```
from django.urls import path
from . import views
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/', views.user_login, name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('register/', views.register, name='register'),
]
```

As the last step, we will create *register.html* and *register_done.html* templates inside the *templates/account* directory. Let's start with *register.html:*

finesauces/accounts/templates/accounts/register.html

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

{% block title %}Register{% endblock %}

{% block content %}
{% if messages %}
  <div class="shadow-custom messages alert alert-danger
    col-lg-5 mx-auto text-center">
    {% for message in messages %}
      {{ message }}
        <a href="" class="close">x</a>
      {% endfor %}
  </div>
{% endif %}
<div class="card shadow-custom border-0 col-lg-5 mx-auto mb-3">
  <h3 class="py-2 font-weight-bold text-grey text-center">
    Register
  </h3>
  <form action="" method="post">
    <div class="form-group">
      {% csrf_token %}
      <div class="input-field">
        {{ user_form.first_name }}
        <label for="first_name">First name</label>
      </div>
      <div class="input-field">
        {{ user_form.last_name }}
        <label for="last_name">Last name</label>
      </div>
      <div class="input-field">
        {{ user_form.email }}
        <label for="email">Email</label>
      </div>
      <div class="input-field">
        {{ user_form.password }}
        <label for="password">Password</label>
      </div>
      <div class="input-field">
        {{ user_form.password2 }}
        <label for="password2">Repeat password</label>
      </div>
    </div>
    <button class="btn btn-block btn-warning mb-3" type="submit">
      Register
    </button>
  </form>
</div>
{% endblock %}
```

To inform user that his account was created, create *register_done.html* template. We can keep it simple:

finesauces/accounts/templates/accounts/register_done.html
```
{% extends 'base.html' %}

{% block title %}Registration Complete{% endblock %}

{% block content %}
<div class="card shadow-custom border-0 col-lg-5 mx-auto mb-3 text-center">
  <h3 class="py-2 font-weight-bold text-grey">
    Welcome, {{ new_user.first_name }}
  </h3>
  <hr class="my-0">
  <div class="py-2">
    Your account was successfully created.
    You can <a href="{% url 'login' %}">log in.</a>
  </div>
</div>
{% endblock %}
```

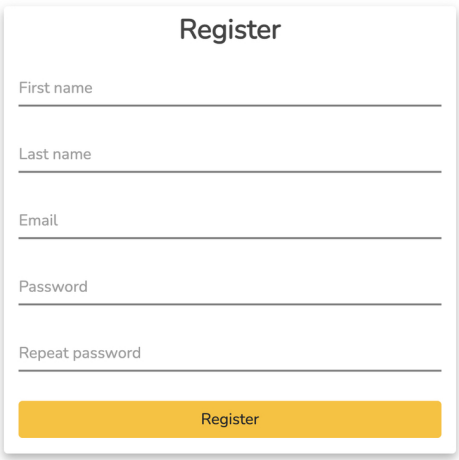
Update *base.html* file to include *register* link in the navbar. Find the following section:

finesauces/listings/templates/base.html
```
#...
<li class="nav-item">
  <a href="" class="nav-link text-white">
    <i class="fas fa-user-plus"></i> Register
  </a>
</li>
#...
```

and adjust it to look like this:

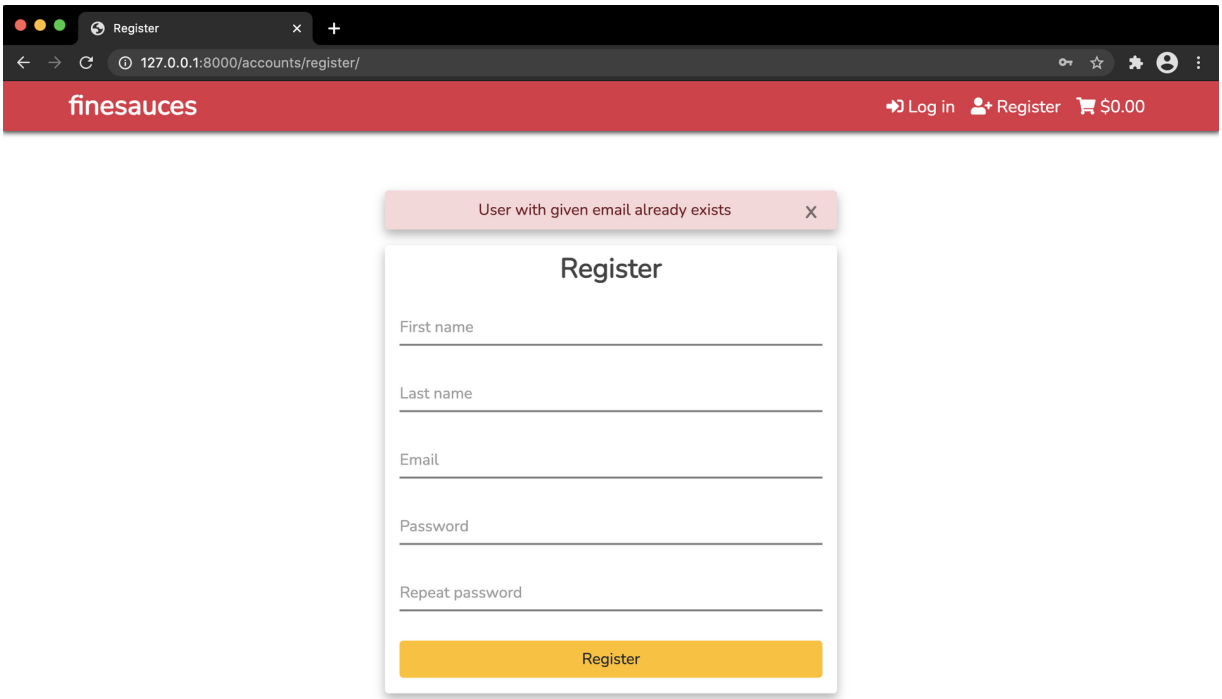
finesauces/listings/templates/base.html
```
#...
<li class="nav-item">
  <a href="{% url 'register' %} " class="nav-link text-white">
    <i class="fas fa-user-plus"></i> Register
  </a>
</li>
#...
```

Let's try it out. Access http://127.0.0.1:8000/accounts/register/ and register a new user:

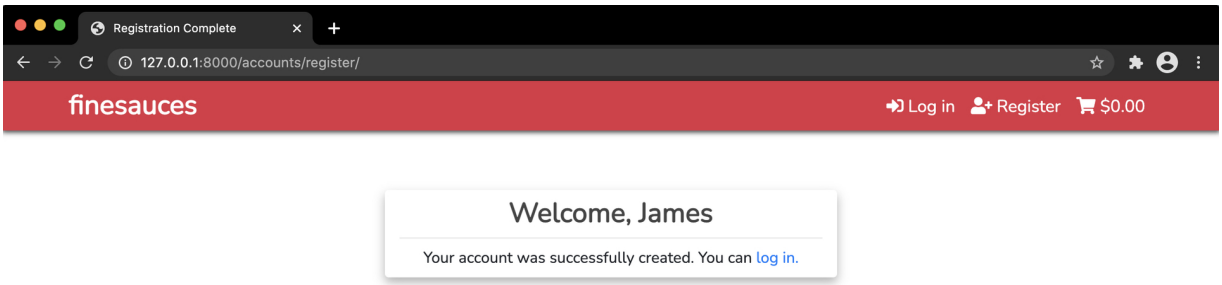


Now our customers are able to register, login, and logout!

If you try to register and the provided email is already taken, you will be notified:



After successful registration, the *register_done.html* template will be rendered:



7.3.1 Updating product review functionality

Back in the *Creating listings application* chapter, we created categories and products for our e-commerce app. We also implemented review functionality, so the users are able to leave their rating and feedback for the products. By default, however, the author's first name is set to "*Anonymous*". Now that customers can create their account and provide their name, it would be great to show that name instead of "*Anonymous*". To make this happen, we will have to return to the *listings* application and modify the *views.py* file.

Find the following section:

```
finesauces/listings/views.py
#...
if review_form.is_valid():
    cf = review_form.cleaned_data

    author_name = "Anonymous"
    Review.objects.create(
        product = product,
        author = author_name,
        rating = cf['rating'],
        text = cf['text']
    )
    #...
```

And update it to include this user check:

```
finesauces/listings/views.py
#...
if review_form.is_valid():
    cf = review_form.cleaned_data

    author_name = "Anonymous"
    if request.user.is_authenticated and request.user.first_name != '':
        author_name = request.user.first_name

    Review.objects.create(
        product = product,
        author = author_name,
        rating = cf['rating'],
        text = cf['text']
    )
    #...
```

We added a simple check to see whether the user writing the review is logged in and has provided his first name. If so, use it instead of default "*Anonymous*".

7.4 Password change views

We want users to be able to change or reset their password if required. Django authentication framework comes with views for this specific purpose. The whole password change process requires only two views and templates. To enable a user to enter new credentials and to confirm the update.

Add the following URL patterns to the *urls.py* file in the *accounts* application directory:

finesauces/accounts/urls.py

```
#...
path('password_change/', auth_views.PasswordChangeView.as_view(),
    name='password_change'),
path('password_change/done/', auth_views.PasswordChangeDoneView.as_view(),
    name='password_change_done'),
```

PasswordChangeView will handle password change form and allow users to change their password. This view will by default look for *password_change_form.html* template in *registration* directory. Create this file and add the following code to it:

finesauces/accounts/templates/registration/password_change_form.html

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

{% block title %}Change password{% endblock %}

{% block content %}
{% if form.errors %}
  <div class="shadow-custom messages alert alert-danger
    col-lg-5 mx-auto text-center">
    Could not update password.
    Make sure you provide valid credentials and try again
  </div>
{% endif %}

<div class="card shadow-custom border-0 col-lg-5 mx-auto mb-3">
  <h3 class="py-2 font-weight-bold text-grey text-center">
    Change password
  </h3>
  <form method="post">
    {% csrf_token %}
    <div class="input-field">
      <label for="old_password" class="text-muted">Old password</label>
      <input class="form-control" type="password" name="old_password"
        autofocus="" required="" id="id_old_password">
    </div>
    <div class="input-field">
      <label for="new_password1" class="text-muted">New password</label>
      <input class="form-control" type="password" name="new_password1"
        required="" id="id_new_password1">
    </div>
    <div class="input-field">
      <label for="new_password2" class="text-muted">Confirm password</label>
      <input class="form-control" type="password" name="new_password2"
        required="" id="id_new_password2">
    </div>
    <button class="btn btn-block btn-warning mb-3" type="submit">
      Change password
    </button>
  </form>
</div>
{% endblock content %}
```

Just a short note. We are providing completely custom form built from scratch in this view. Since we made sure names and ids of input fields are consistent with what Django expects to receive, we are able to use it without any issues. If you would like to use default Django form, feel free to replace section between *{% csrf_token %}* and *<button>* with template tag *{{ form.as_p }}*.

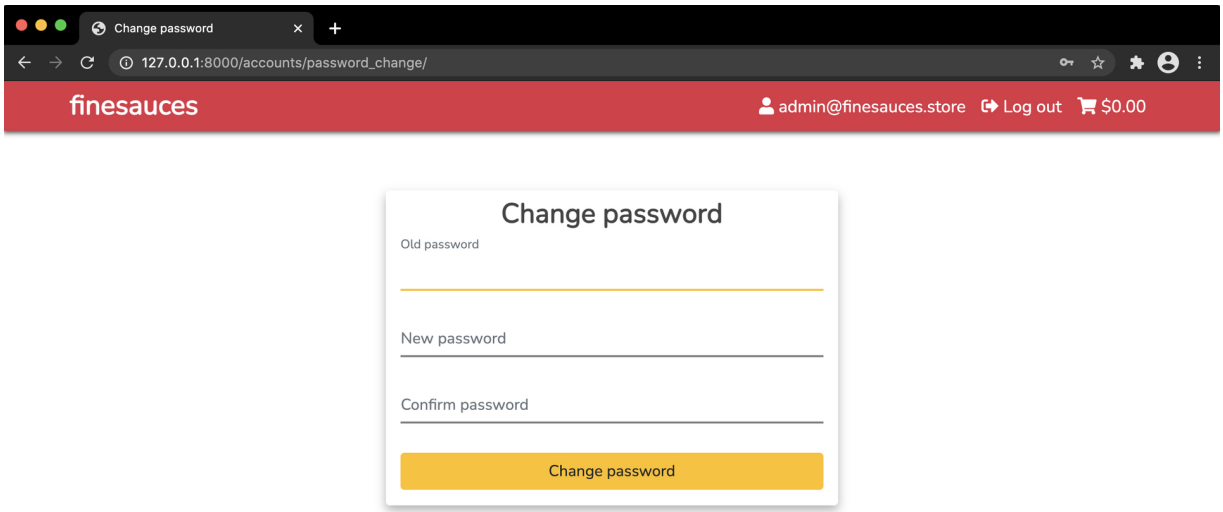
After a successful password change, the *PasswordChangeView,* by default, redirects the user to the *'password_change_done'* URL mapped to *PasswordChangeDoneView* . This view then looks for the *password_change_done.html* template. Create this template in the same folder as the previous template. Add the following code to it:

finesauces/accounts/templates/registration/password_change_done.html

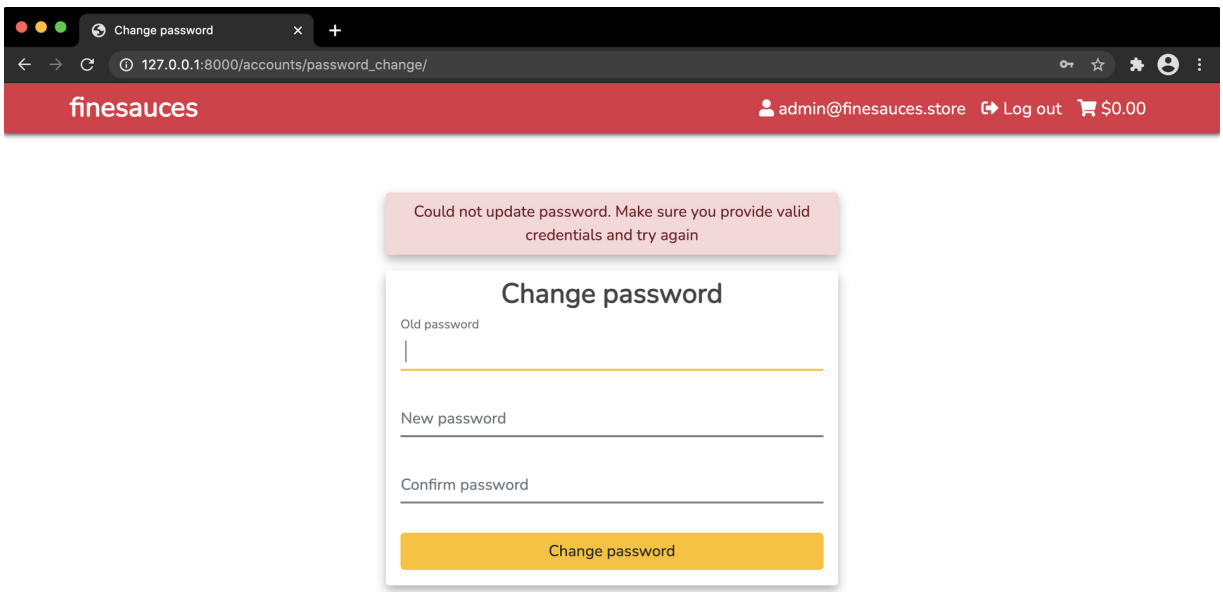
```
{% extends 'base.html' %}

{% block title %}Password changed{% endblock %}

{% block content %}
<div class="card shadow-custom border-0 col-lg-5 mx-auto mb-3 text-center">
  <h3 class="py-2 font-weight-bold text-grey">
    Password changed
  </h3>
  <hr class="my-0">
  <div class="py-2">
    Your password was changed successfully!
  </div>
</div>
```

```
{% endblock %}
```

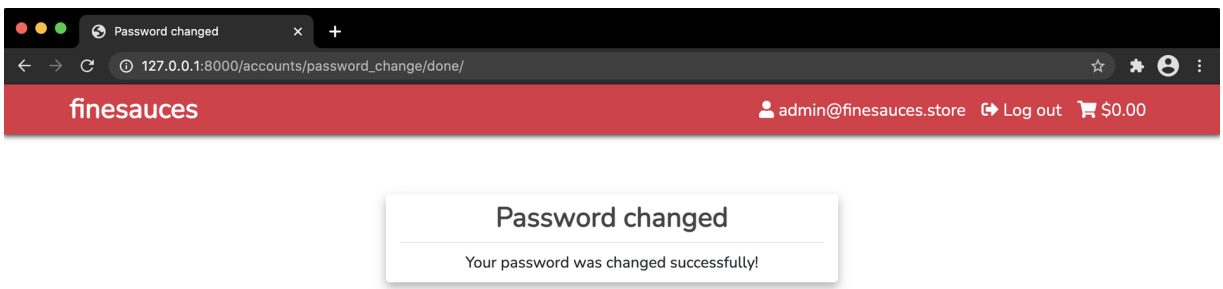
Let's visit http://127.0.0.1:8000/accounts/password_change/ to check our password change form. If you are not logged in, you will be redirected to the login page first (thanks to *LOGIN_URL = 'login'* setting).



If we provide invalid credentials, either incorrect current password or our new passwords don't match, we will receive an error notification, which we defined in the top part of *password_change_form.html* template:



After changing the password successfully, *PasswordChangeView* redirects us to the *'password_change_done'* URL, which is mapped to *PasswordChangeDoneView:*



7.5 Password reset views

Django authentication framework comes with four views for resetting the password.

- ***PasswordResetView*** generates a one-time use link that can be used to reset the password. The link will be sent to the user's registered email address.
- ***PasswordResetDoneView*** displays a page after a user has been emailed a link to reset their password.
- ***PasswordResetConfirmView*** presents a form for entering a new password.
- ***PasswordResetCompleteView*** informs the user that the password has been successfully changed.

Let's start with the *PasswordResetView* >view. This view generates a one-time use link for resetting the password. If the user's email address does not exist in the system, this view won't send an email, but the user won't receive any error message either. This prevents the information from leaking to potential attackers.

Open *urls.py* file in *account* application and add the following URL to the list:

finesauces/accounts/urls.py

```
#...
path('password_reset/', auth_views.PasswordResetView.as_view(),
    name='password_reset'),
```

This view looks for *password_reset_form.html* file inside *registration* folder. Let's create this file and add the following code to it:

finesauces/accounts/templates/registration/password_reset_form.html

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

{% block title %}Reset password{% endblock %}

{% block content %}
<div class="card shadow-custom border-0 col-lg-5 mx-auto mb-3">
  <h3 class="py-2 font-weight-bold text-grey text-center">
    Forgot your password?
  </h3>
  <p class="text-center">
    Your email address for sending password reset instructions
  </p>
  <form method="post">
    {% csrf_token %}
    <div class="input-field">
      <label for="email" class="text-muted">Email</label>
      <input class="form-control" type="email" name="email"
        required="" id="id_email">
    </div>
    <button class="btn btn-block btn-warning mb-3" type="submit">
      Send email
    </button>
  </form>
</div>
{% endblock %}
```

Once the reset form is successfully submitted, this view will send user an email with link to reset the password. This email template is by default expected to be inside *password_reset_email.html* file. Create this file and add the following code to it:

finesauces/accounts/templates/registration/password_reset_email.html

```
You have requested password reset for email {{ email }}.
Follow this link to reset your password:
{{ protocol }}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid
token=token %}
```

Email template context:

- ***email*** - an alias for user.email
- ***protocol*** - http or https
- ***domain*** - address of our site
- ***url*** '***password_reset_confirm***' - this URL redirects us to the view with form for entering new password
- ***uid*** - user's primary key encoded in base 64
- ***token*** - token to check that the reset link is valid

PasswordResetDone view is called when user has been emailed a link to reset their password. Let's add this view to the list of our URLs.

finesauces/accounts/urls.py

```
#...
path('password_reset/done/', auth_views.PasswordResetDoneView.as_view(),
    name='password_reset_done'),
```

We need to provide *password_reset_done.html* template for this view. Place the following code inside this file:

finesauces/accounts/templates/registration/password_reset_done.html

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

{% block title %}Password reset{% endblock %}

{% block content %}
<div class="card shadow-custom border-0 col-lg-5 mx-auto mb-3 text-center">
  <h3 class="py-2 font-weight-bold text-grey">
    Password reset
  </h3>
  <hr class="my-0">
  <div class="py-2">
    We have sent you instructions for resetting your password.
    In case you haven't received any email,
    please make sure you provided correct email address.
  </div>
</div>
{% endblock %}
```

PasswordResetConfirmView renders a form for entering a new password. Keyword arguments from the URL are *uidb64* and *token*. First, let's add this view to the URL list. Open the *urls.py* file and add the following link to the list:

finesauces/accounts/urls.py
```
#...
path('password_reset/<uidb64>/<token>/',
    auth_views.PasswordResetConfirmView.as_view(),
    name='password_reset_confirm'),
```

Template for this view is called *password_reset_confirm.html*. Fill in the following code:

finesauces/accounts/templates/registration/password_reset_confirm.html
```
{% extends 'base.html' %}

{% block title %}Reset password{% endblock %}

{% block content %}
{% if form.errors %}
  <div class="shadow-custom messages alert alert-danger
    col-lg-5 mx-auto text-center">
    Could not update password.
    Make sure provided passwords match and try again.
  </div>
{% endif %}

<div class="card shadow-custom border-0 col-lg-5 mx-auto mb-3">
  <h3 class="py-2 font-weight-bold text-grey text-center">
    Reset password
  </h3>
  {% if validlink %}
    <p class="text-center">Fill in new password</p>
    <form method="post">
      {% csrf_token %}
      <div class="input-field">
        <label for="id_new_password1">New password</label>
        <input class="form-control" type="password" name="new_password1"
        required="" id="id_new_password1">
      </div>
       <div class="input-field">
        <label for="id_new_password2">
          Confirm new password
        </label>
        <input class="form-control" type="password" name="new_password2"
        required="" id="id_new_password2">
      </div>
      <button class="btn btn-block btn-warning mb-3" type="submit">
        Confirm password
      </button>
    </form>
  {% else %}
    <p>
      Provided link is invalid.
      Please request new password reset.
    </p>
  {% endif %}
</div>
{% endblock %}
```

Variable *validlink* is a Boolean value, *True* if the link (combination of *uidb64* and *token*) is valid and unused.

After setting new password, we are redirected to *'password_reset_view'* URL. Let's map this URL to the *PasswordReserCompleteView*. Add the following pattern to the *urls.py* file:

finesauces/accounts/urls.py
```
#...
path('password_reset/complete/',
    auth_views.PasswordResetCompleteView.as_view(),
    name='password_reset_complete'),
```

Create new template *password_reset_complete.html* and type in the following code:

finesauces/accounts/templates/registration/password_reset_complete.html
```
{% extends 'base.html' %}

{% block title %}Password changed{% endblock %}

{% block content %}
<div class="card shadow-custom border-0 col-lg-5 mx-auto mb-3
  text-center">
  <h3 class="py-2 font-weight-bold text-grey">
    Password changed
  </h3>
  <hr class="my-0">
  <div class="py-2">
    Your password was successfully changed. You can
    <a class="text-decoration-none" href="{% url 'login' %}">
```

```
      log in.
    </a>
  </div>
</div>
{% endblock %}
```

Your *templates* folder inside *accounts* application should have the following structure:

```
templates
├── account
│   ├── login.html
│   ├── register.html
│   └── register_done.html
└── registration
    ├── password_change_done.html
    ├── password_change_form.html
    ├── password_reset_complete.html
    ├── password_reset_confirm.html
    ├── password_reset_done.html
    ├── password_reset_email.html
    └── password_reset_form.html
```

Your *urls.py* file should look similar to this:

finesauces/accounts/urls.py

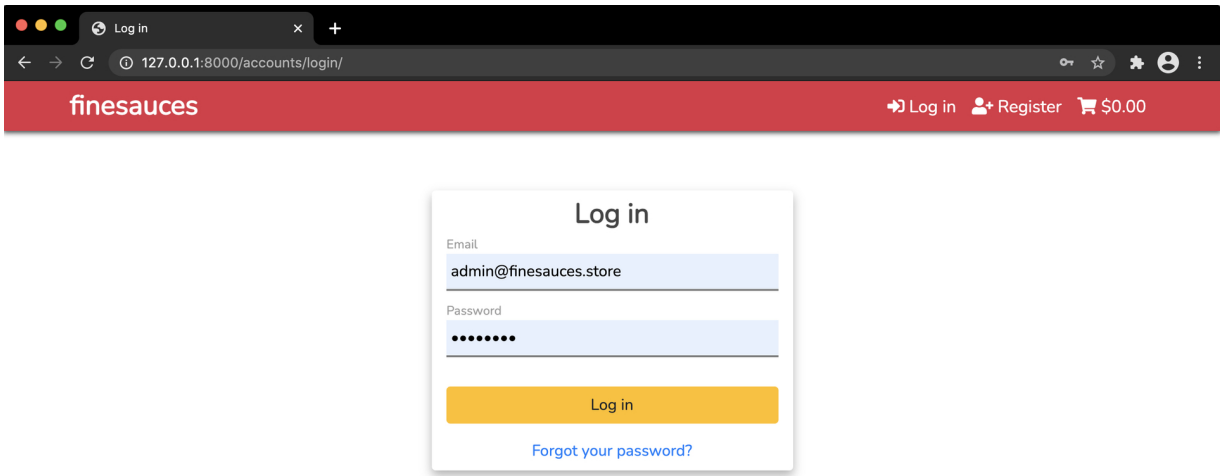
```
from django.urls import path
from . import views
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/', views.user_login, name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('register/', views.register, name='register'),

    # password change
    path('password_change/', auth_views.PasswordChangeView.as_view(),
        name='password_change'),
    path('password_change/done/', auth_views.PasswordChangeDoneView.as_view(),
        name='password_change_done'),

    #password reset
    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('password_reset/<uidb64>/<token>/',
        auth_views.PasswordResetConfirmView.as_view(),
        name='password_reset_confirm'),
    path('password_reset/complete/',
        auth_views.PasswordResetCompleteView.as_view(),
        name='password_reset_complete'),
]
```

To wrap up this section, let's update our *login.html* template to include a password reset link. Add the following code after closing *</form>* tag:

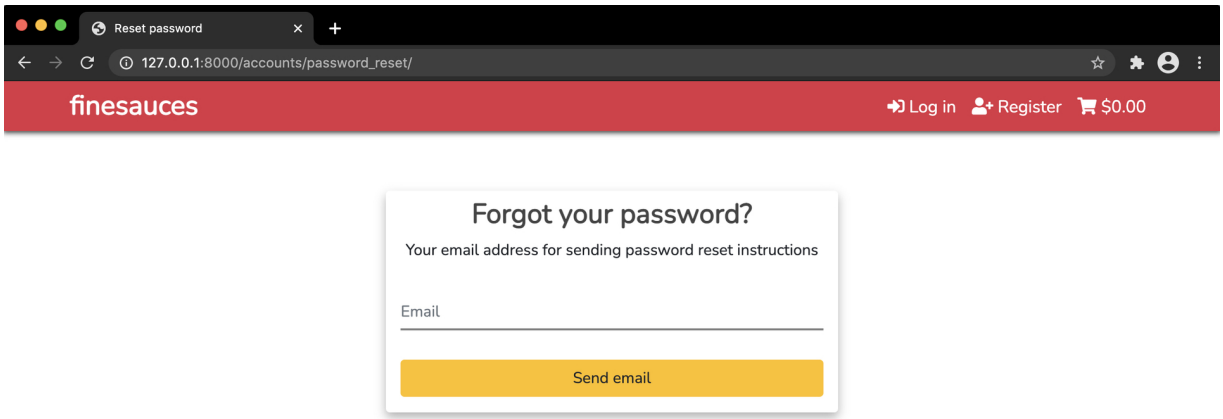
finesauces/accounts/templates/accounts/login.html

```
#...
<div class="text-center mb-2">
  <a href="{% url 'password_reset' %}"
    class="text-decoration-none">
    Forgot your password?
  </a>
</div>
#...
```

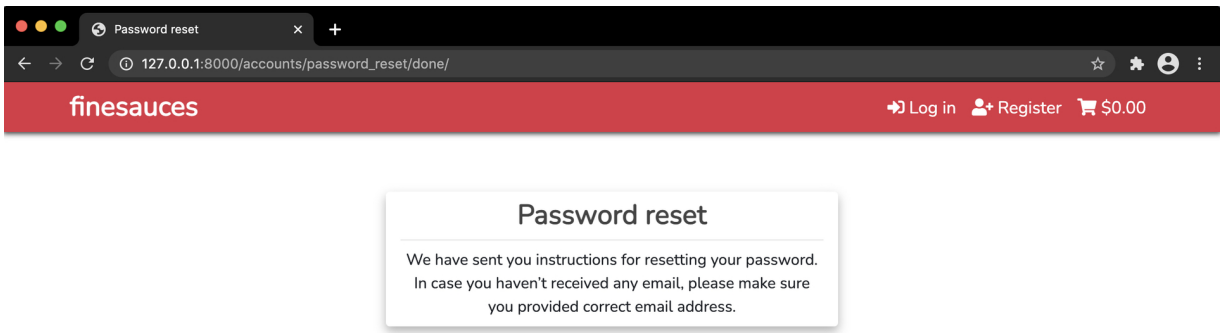
It's time to test our password reset process. Let's make sure our development server is running and visit http://127.0.0.1:8000/accounts/login/ :



and click on *Forgot your password* link. We will be redirected to our *PasswordResetView* form to provide our email address:



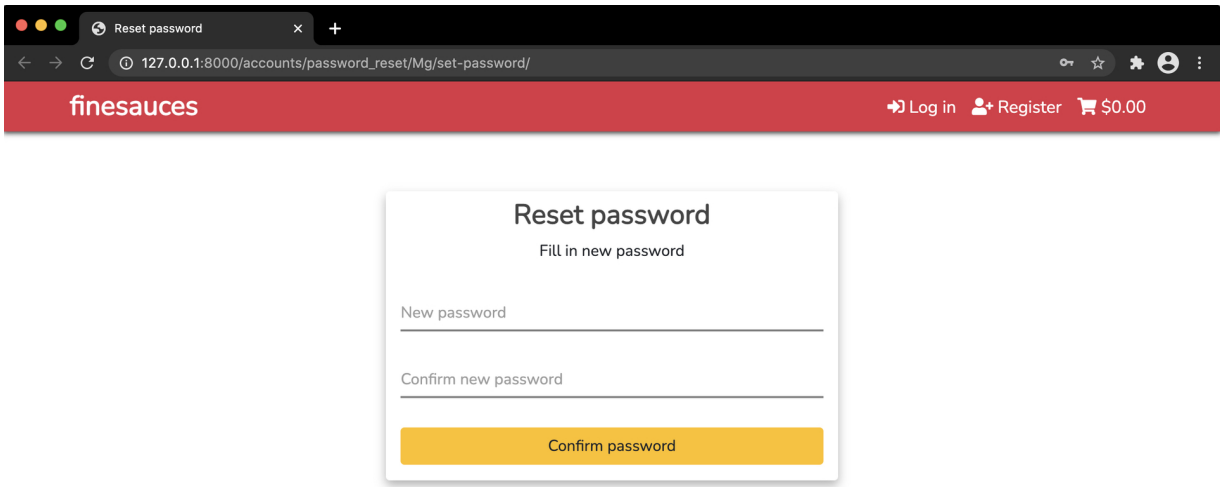
Once we provide a valid email address for sending password reset instructions, we are redirected to *PasswordResetDone* view:



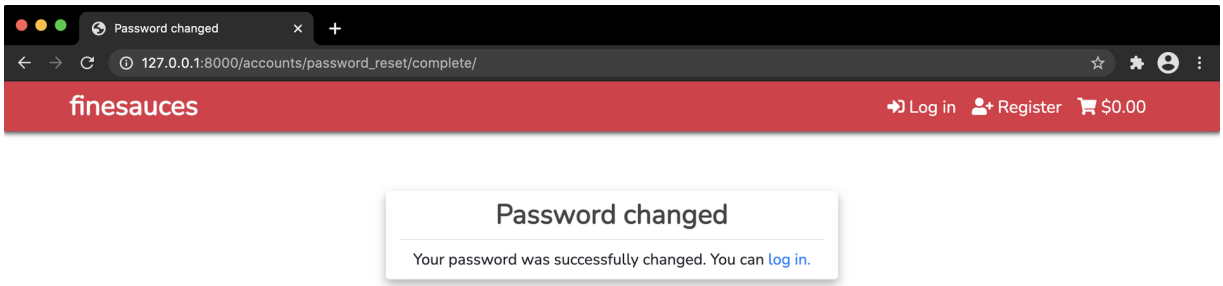
Django then sends us an email containing a password reset link:

```
You have requested password reset for email admin@finesauces.store.
Follow this link to reset your password:
http://127.0.0.1:8000/account/password_reset/Nw/ac9iak-efa5d476ec317fb6d5162e93d5f9c30e/
```

Clicking on the provided link sends us to the *PasswordResetConfirmView* to provide a new password:



After successfully resetting our password, we receive the following confirmation from *PasswordResetCompleteView:*



Our users can now log in, log out, register, change, and reset their password!

7.6 Extending User model

So far, the default *User* model provided by Django was just enough for our needs. However, when working on bigger projects, you will often find the default fields provided by the *User* model insufficient.

We can resolve this limitation by either creating a custom user model [56] or extending our default *User* model by implementing a new *Profile* model and connecting them by using a one-to-one relationship. The latter approach is pretty straightforward and offers a bit more flexibility compared to creating a custom user model. However, the custom user model removes the need for additional or complex database queries to retrieve related models. Each approach has its pros and cons.

For this project, we will go with creating a *Profile* model to store our additional information. Edit the *models.py* file of the *accounts* application and add the following code to it:

finesauces/accounts/models.py

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

class Profile(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE
)
    phone_number = models.CharField(max_length=20, blank=True)
    address = models.CharField(max_length=250, blank=True)
    postal_code = models.CharField(max_length=20, blank=True)
    city = models.CharField(max_length=100, blank=True)
    country = models.CharField(max_length=100, blank=True)

    def __str__(self):
        return f'{self.user.username} profile'
```

We define a one-to-one relationship between *User* and *Profile* models, meaning one user can have only one profile and one profile can belong only to one user. As recommended by Django documentation [57] , we reference the *User* model by the *AUTH_USER_MODEL* setting.

Let's create initial migration for our *Profile* model:

terminal

```
(env) ~/finesauces$ python manage.py makemigrations
```

Run the *migrate* command to get our database up to date with our new model:

terminal

```
(env) ~/finesauces$ python manage.py migrate
```

Now we need to add our *Profile* model to the admin site. Open *admin.py* file inside the *account* directory and add the following code to it:

finesauces/accounts/admin.py

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

@admin.register(Profile)
class ProfileAdmin(admin.ModelAdmin):
    list_display = ['user', 'phone_number', 'address', 'postal_code', 'city']
```

When a customer creates a new account, we also want to create a blank *Profile* object for this customer. Open the *views.py file* of the *account* application, add the import for our *Profile* model and code for creating an empty profile:

finesauces/accounts/views.py

```
from .models import Profile
#...
# Create a new user object
new_user = User.objects.create_user(
   #...
)

# Create the user profile
Profile.objects.create(user=new_user)
```

```
return render(
#...
```

7.7 Creating user profile section

We have successfully extended our basic *User* model by implementing *Profile* model. As for next step, let's create a customer profile section. We want to show user information there. This means showing data from both *User* and *Profile* models. User will be also able to edit the account information. We will create two forms to handle this functionality. Open *forms.py* file in *accounts* directory and add the following code:

finesauces/accounts/forms.py

```
from .models import Profile

class UpdateUserForm(forms.ModelForm):

    class Meta:
        model = User
        fields = ('first_name', 'last_name', 'email')

        widgets = {
            'first_name': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'last_name': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'email': forms.EmailInput(
                attrs={'class': 'form-control'}
            )
        }
class UpdateProfileForm(forms.ModelForm):

    class Meta:
        model = Profile
        fields = ('phone_number', 'address', 'postal_code', 'city', 'country')

        widgets = {
            'phone_number': forms.TextInput(
                attrs={'class': 'form-control'}
            ),
            'address': forms.TextInput(
                 attrs={'class': 'form-control'}
            ),
            'postal_code': forms. TextInput (
                attrs={'class': 'form-control'}
            ),
            'city': forms. TextInput (
                attrs={'class': 'form-control'}
            ),
            'country': forms. TextInput (
                attrs={'class': 'form-control'}
            )
        }
```

As for the view, open the *views.py* file inside the *accounts* application and add the following code to it:

finesauces/accounts/views.py

```
#...
from .forms import (
    LoginForm,
    UserRegistrationForm,
    UpdateUserForm,
    UpdateProfileForm
)
from django.contrib.auth.decorators import login_required

@login_required
def profile(request):
    if request.method == 'POST':
        email = request.POST['email']
        user = None

        try:
            user = User.objects.get(email=email)
        except User.DoesNotExist:
            pass

        if user is None or user.id == request.user.id:
            user_form = UpdateUserForm(
                instance=request.user,
                data=request.POST
             )
            profile_form = UpdateProfileForm(
                instance=request.user.profile,
                data=request.POST,
            )
```

```
            if user_form.is_valid() and profile_form.is_valid():
                user_form.save()
                profile_form.save()

                messages.success(request, 'Profile was updated successfully')
            else:
                messages.error(request, 'User with given email already exists')

            return redirect('profile')

        else:
            user_form = UpdateUserForm(instance=request.user)
            profile_form = UpdateProfileForm(instance=request.user.profile)

        return render(
            request,
            'accounts/profile.html',
            {
                'user_form': user_form,
                'profile_form': profile_form
            }
        )
#...
```

We start by decorating our *profile* view with @*login_required* decorator, which requires a user to be logged in. If a user is not logged in, this view redirects visitor to the *login* page.

When a user visits the profile page, we render an *accounts/profile.html* template and provide a profile form responsible for handling our user data. This form is provided with the instance of existing *User* and *Profile* models that are supposed to be displayed and updated if required.

When our form is submitted via a *POST* request, we first check if a user with the given email already exists. If such user already exists, we check if it's us. If so, we then proceed and save our forms with associated *User* and *Profile* instances.

Now we need to create a template for showing our profile page. Go ahead and create a *profile.html* template inside the *templates/accounts* directory within the *accounts* application. Add the following code to it:

finesauces/accounts/templates/accounts/profile.html

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

{% block title %}Account{% endblock %}

{% block content %}
{% if messages %}
  {% for message in messages %}
    {% if message.tags == 'error' %}
    <div class="shadow-custom messages alert alert-danger text-center
      col-lg-7 mx-auto">
    {% else %}
    <div class="shadow-custom messages alert alert-success text-center
      col-lg-7 mx-auto">
    {% endif %}
      {{ message }}
      <a href="" class="close">x</a>
    </div>
  {% endfor %}
{% endif %}
  <div class="card shadow-custom border-0 col-lg-7 mx-auto mb-3">
    <h2 class="font-weight-bold text-grey mt-2">Profile</h2>
    <form action="" method="POST">
      {% csrf_token %}
      <div class="row">
        <div class="col-md-6">
          <div class="input-field">
            {{ user_form.first_name }}
            <label for="first_name">First name</label>
          </div>
        </div>
        <div class="col-md-6">
          <div class="input-field">
             {{ user_form.last_name }}
            <label for="last_name">Last name</label>
          </div>
        </div>
        <div class="col-md-6">
          <div class="input-field">
            {{ user_form.email }}
            <label for="email">Email</label>
          </div>
        </div>
        <div class="col-md-6">
          <div class="input-field">
            {{ profile_form.phone_number }}
            <label for="phone_number">Phone number</label>
          </div>
        </div>
      </div>
      <hr class="mt-0">
```

```
      <div class="row">
        <div class="col-md-6">
          <div class="input-field">
            {{ profile_form.address }}
            <label for="address">Address</label>
          </div>
        </div>
        <div class="col-md-6">
          <div class="input-field">
            {{ profile_form.postal_code }}
            <label for="postal_code">Postal code</label>
          </div>
        </div>
        <div class="col-md-6">
          <div class="input-field">
            {{ profile_form.city }}
            <label for="city">City</label>
          </div>
        </div>
        <div class="col-md-6">
          <div class="input-field">
            {{ profile_form.country }}
            <label for="country">Country</label>
          </div>
         </div>
      </div>
      <hr>
      <div class="row">
        <button type="submit" class="btn col" data-toggle="tooltip"
          data-placement="top"
          title="Fill in new details and click 'Update profile'">
          Update profile</button>
        <a class="btn col" href="{% url 'password_change' %}">
          Change password
        </a>
      </div>
      <hr>
    </form>
  </div>
</div>
{% endblock %}
```

Now we need to modify *urls.py* file in the *account* directory and add the following URL pattern to the file:

finesauces/accounts/urls.py

```
#...
path('profile/', views.profile, name='profile'),
#...
```

And finally, add our *profile* URL to the navbar inside the *base.html* template. Find the following code:

finesauces/listings/templates/base.html

```
#...
<li class="nav-item">
  <a href="" class="nav-link text-white">
    <i class="fas fa-user"></i> {{ request.user.email }}
  </a>
</li>
#...
```

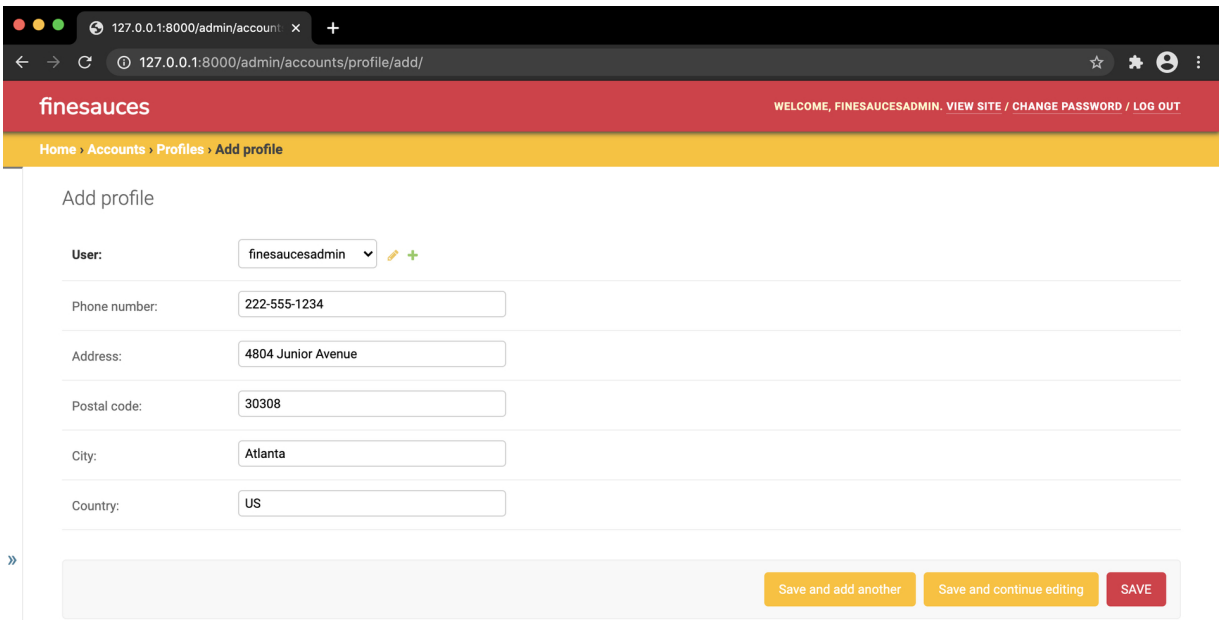
and update it to include our *profile* URL:

finesauces/listings/templates/base.html

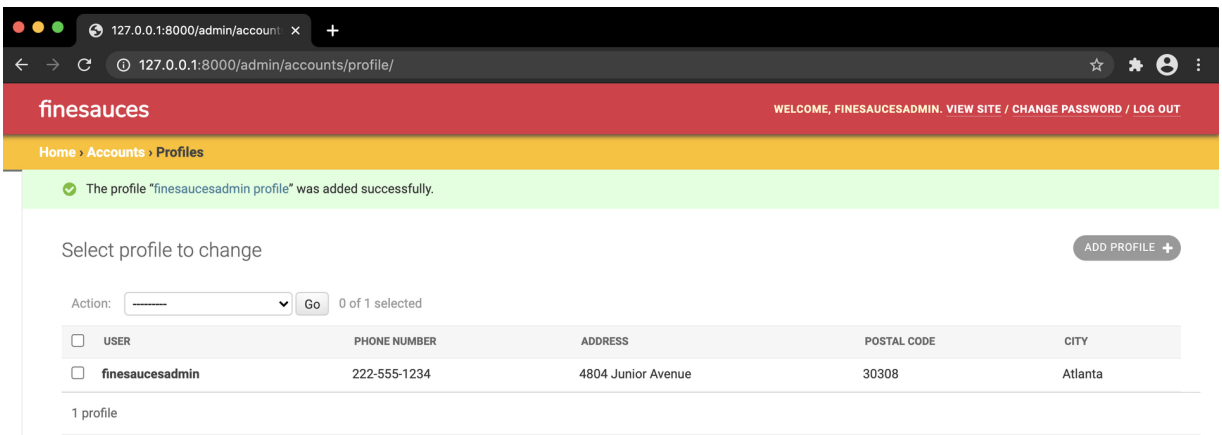
```
#...
<li class="nav-item">
  <a href="{% url 'profile' %} " class="nav-link text-white">
    <i class="fas fa-user"></i> {{ request.user.email }}
  </a>
</li>
#...
```

Make sure the local development server is running and visit http://127.0.0.1:8000/ . The email address in the navbar will contain a link to the user profile. Go ahead and click on that link. You will see:

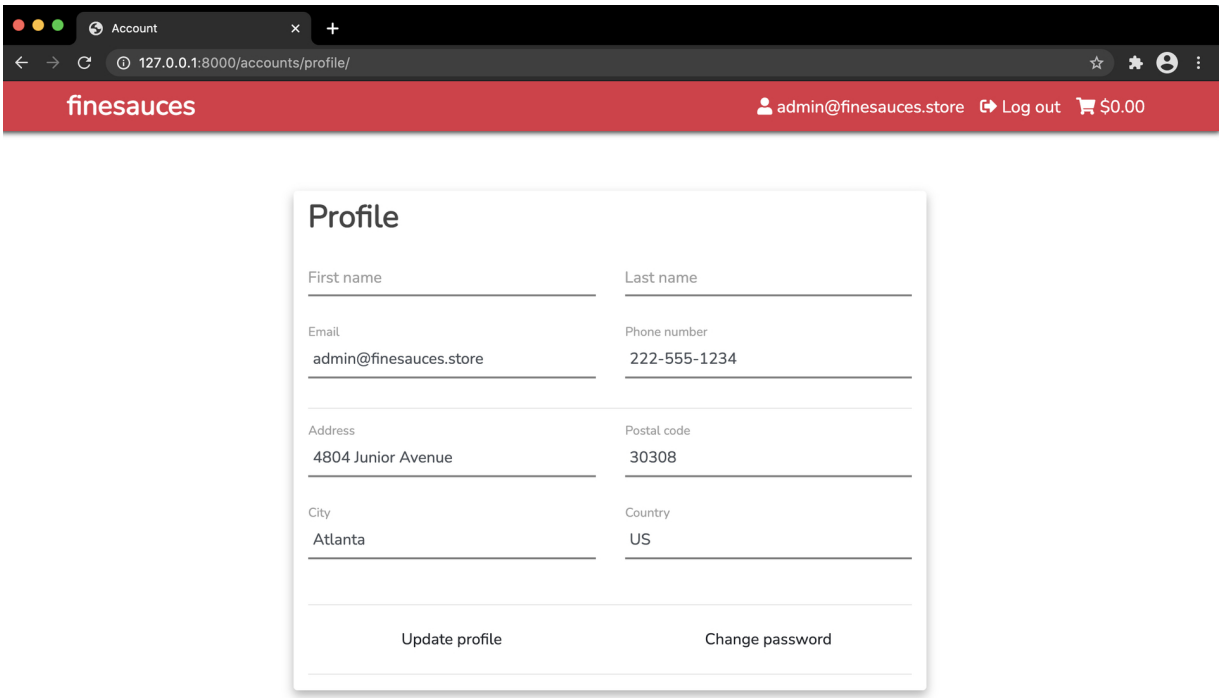
We are greeted with the exception *RelatedObjectDoesNotExist*. Well, what does that mean? Check the exception value below that; *User has no profile*. Remember that our associated profile is created upon user creation. We have created our superuser at the beginning of the project, long before our *Profile* model. Let's remedy this issue. Open http://127.0.0.1:8000/admin/accounts/profile/add/ and create *Profile* record manually:



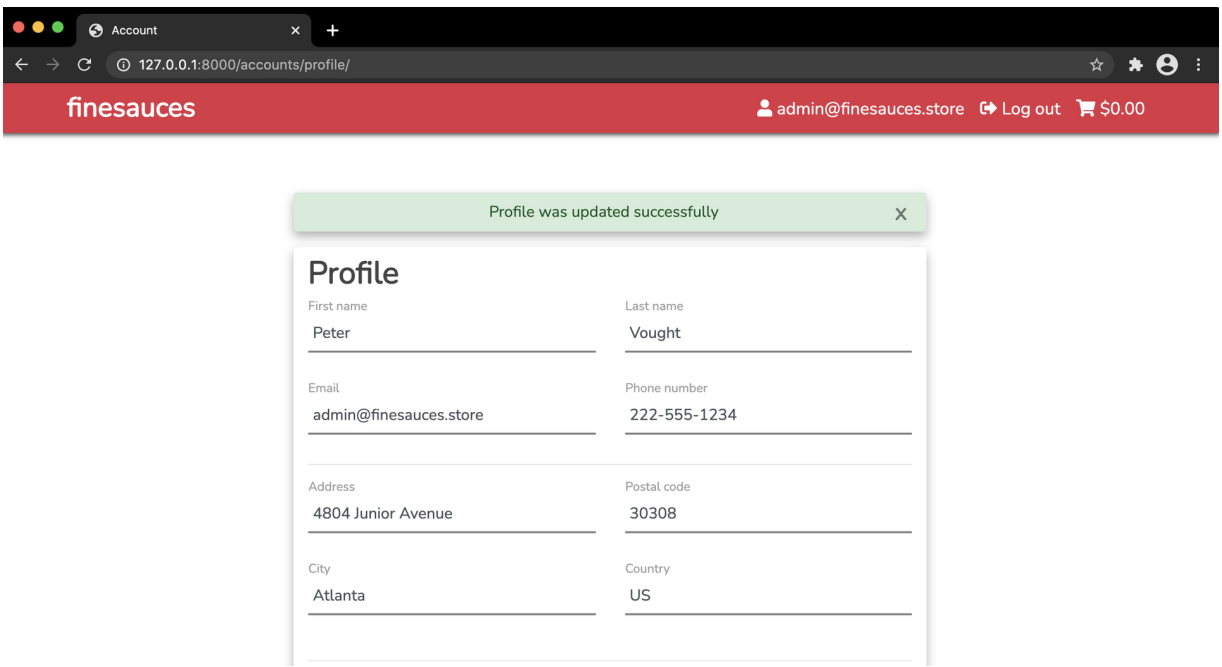
Select a user from the dropdown, fill in required data, and click *on Save:*



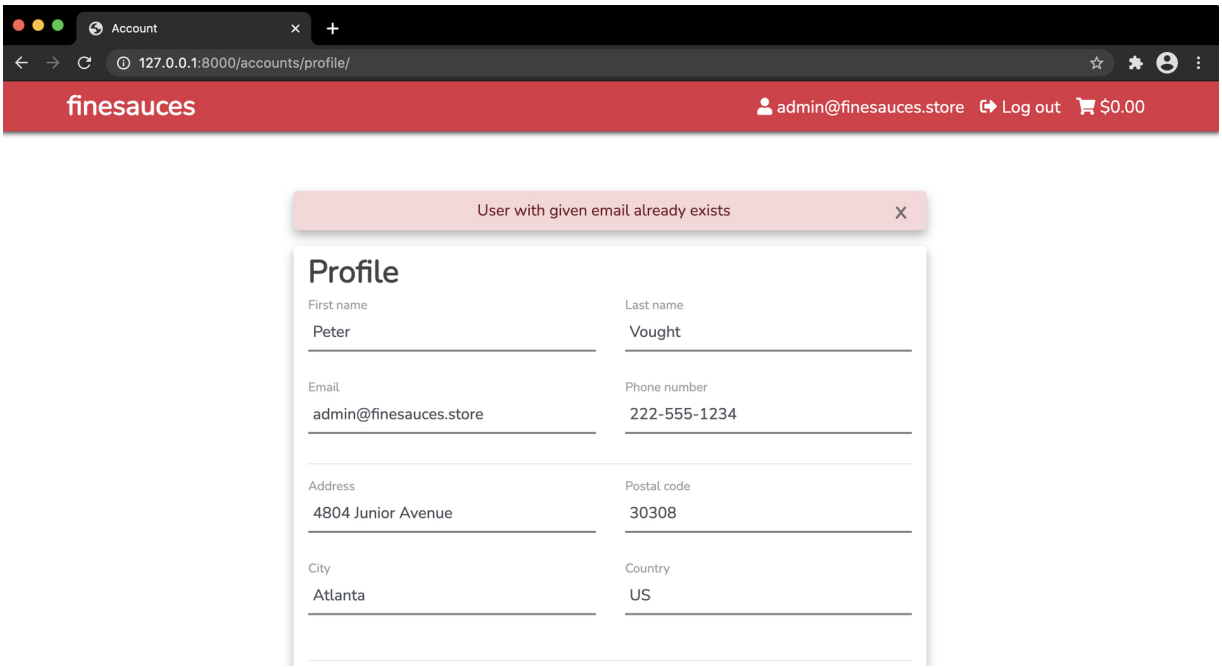
Now let's try to access the profile page http://127.0.0.1:8000/accounts/profile/ again. We are greeted with a profile page. It contains all *User* and *Profile* information along with an *Update profile* and *Change password* links:



Let's test our *Update profile* functionality. Fill in some new data and click on the *Update profile* link. After the profile update, we are also notified by a success message:



If we tried to update the email address, which is already in our database and used by another user, we would receive a following error message:



7.8 Linking orders to customers

When a customer wants to check out the shopping cart and place an order, it would be helpful to prefill their Delivery information form with available user information if a customer is logged in.

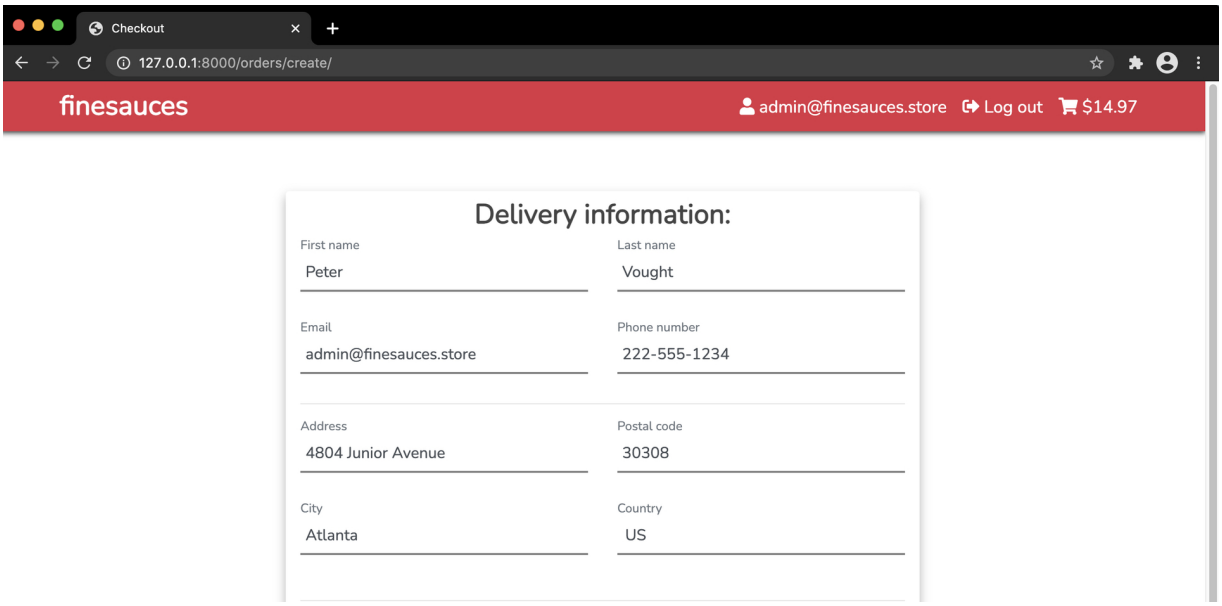
Open the *views.py* file inside the *orders* application and find the *else* statement in the *order_create* view:

```
finesauces/orders/views.py
#...
else:
    order_form = OrderCreateForm()
    #...
```

Update it to look like this:

```
finesauces/orders/views.py
#...
else:
    order_form = OrderCreateForm()
    if request.user.is_authenticated:
        initial_data = {
             'first_name': request.user.first_name,
            'last_name': request.user.last_name,
            'email': request.user.email,
            'telephone': request.user.profile.phone_number,
            'address': request.user.profile.address,
            'postal_code': request.user.profile.postal_code,
            'city': request.user.profile.city,
            'country': request.user.profile.country,
        }
        order_form = OrderCreateForm(initial=initial_data)
        #...
```

If a user is logged in, we prefill the order form with available data via the *initial* attribute. If we were to check out the shopping cart while logged in, we would see our *Delivery information* form prefilled with available user and profile data:



To assign the created order to the specific user, we need to update the *Order* model. Open *models.py* file inside *orders* application and add the following code to it:

finesauces/orders/models.py

```
#...
from django.conf import settings
#...
class Order(models.Model):
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='orders',
        blank=True,
        null=True
    )
    #...
```

In order to be able to create orders for not authenticated users, that is when we have no specific user object to reference, we need to add *blank* and *null* attributes:

- ***blank*** field is form validation-related. If *blank=True, the* Django form will allow an empty value.
- ***null*** is database-related. When set to *True* , it can be saved in a database table as a *NULL* value.

Let's create a migration for our updated *Order* model:

terminal

```
(env) ~/finesauces$ python manage.py makemigrations
```

Now we have to sync the database with our updated model by running *migrate* command:

terminal

```
(env) ~/finesauces$ python manage.py migrate
```

Let's also update our *order_create* view inside the *views.py* file in the *orders* application. Add the following code to this view:

finesauces/orders/views.py

```
#...
order = order_form.save(commit=False)
if request.user.is_authenticated:
    order.user = request.user
order.transport_cost = Decimal(transport_cost)
order.save()
#...
```

Before creating an order record, we check if a user is authenticated to save user reference. Otherwise, we save the *NULL* value to the table. When an anonymous user tries to create a new order, we could offer an option to log in. Let's update the *create.html* template inside the *orders* application. Find the following section:

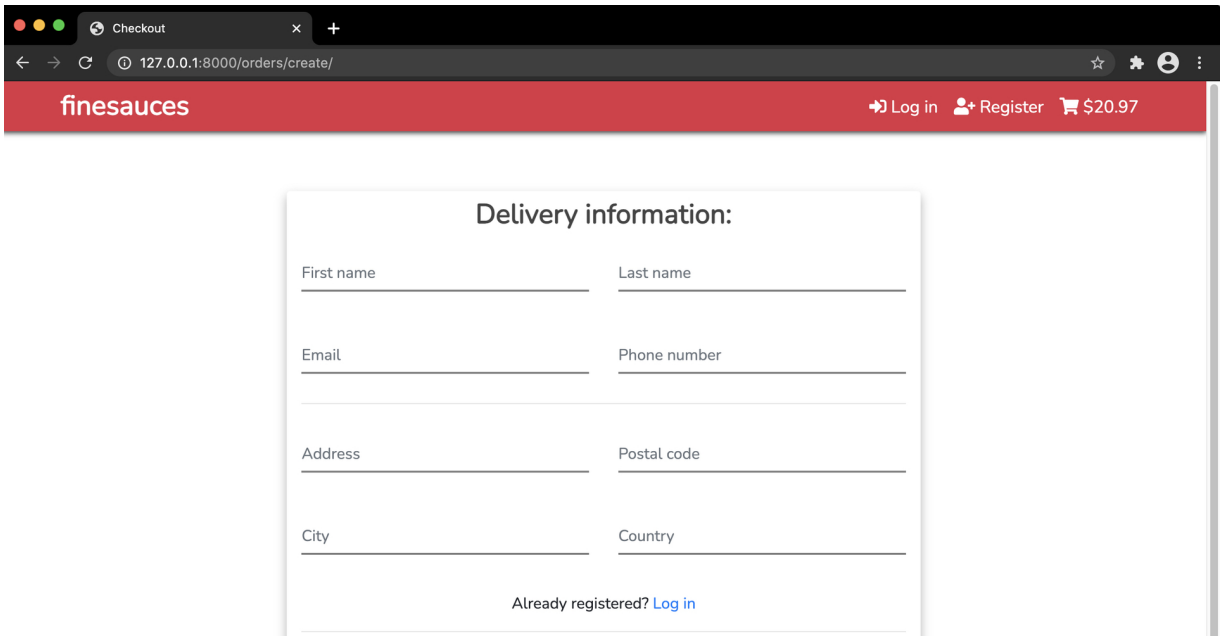
finesauces/orders/templates/order_create.html

```
#...
</div>
<hr>
<h4 class="py-2 font-weight-bold text-grey">
  Transport:
</h4>
#...
```

and add the following code between </*div*> and <*hr*> tags:

```
finesauces/orders/templates/order_create.html
#...
</div>
{% if not request.user.is_authenticated %}
  <div class="mt-2 text-center">
    Already registered?
    <a href="{% url 'login' %}"
      class="text-decoration-none">
      Log in
    </a>
  </div>
{% endif %}
<hr>
#...
```

During checkout, an anonymous user is now offered the option to log in:



7.9 Displaying customer orders

Let's work on displaying orders on the profile page. Open the *profile.html* template in the *account* application and find the following code at the bottom:

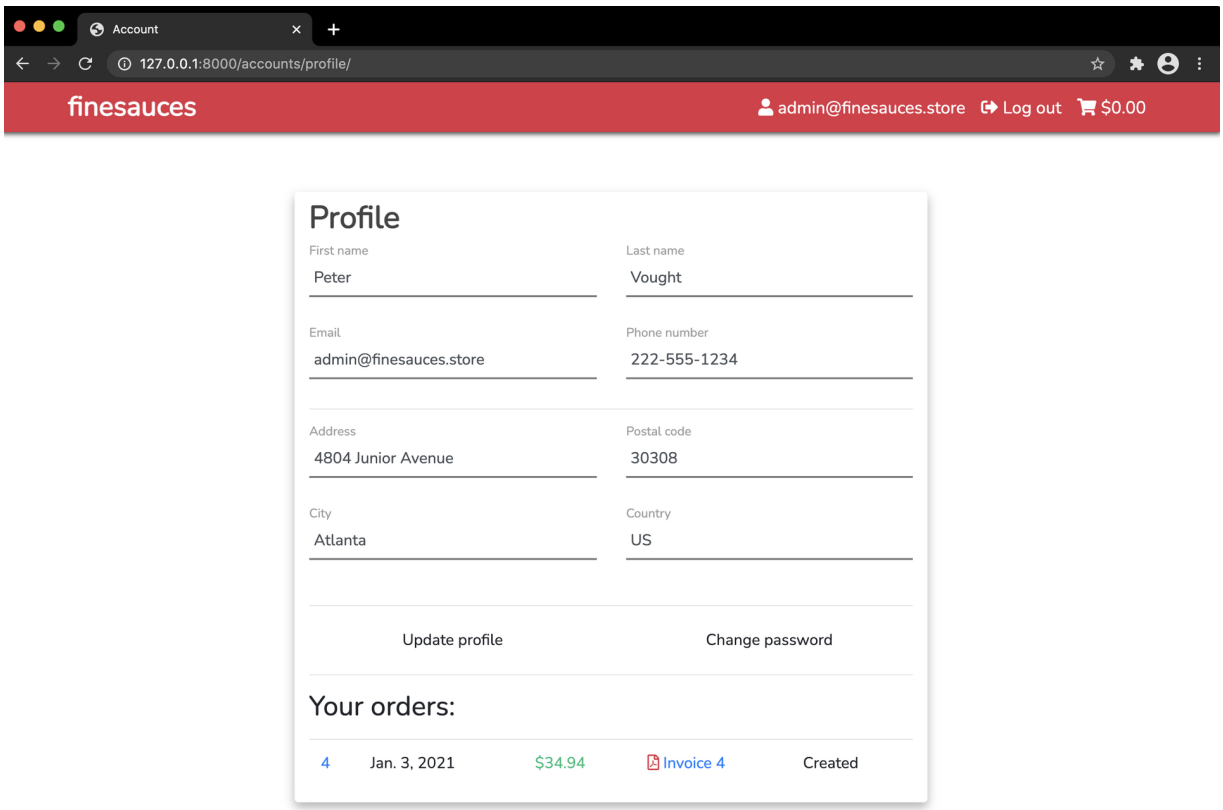
```
finesauces/accounts/templates/accounts/profile.html
        #...
        <a class="btn col" href="{% url 'password_change' %}">
          Change password
        </a>
      </div>
      <hr>
    </form>
  </div>
</div>
{% endblock %}
```

Insert the following code between <*hr*> and </*form*> tag:

```
finesauces/accounts/templates/accounts/profile.html
#...
<h3 class="mb-3">Your orders:</h3>
<table class="table">
  {% for order in request.user.orders.all %}
    <tr>
      <td>
        <a href="" class="text-decoration-none">
          {{ order.id }}
        </a>
      </td>
      <td>{{ order.created|date }}</td>
      <td class="text-green">
        ${{ order.get_total_cost|floatformat:2 }}
      </td>
      <td>
        <a href="" class="text-decoration-none" target="_blank">
          <i class="far fa-file-pdf text-danger"></i>
          Invoice {{ order.id }}
        </a>
      </td>
      <td>{{ order.status }}</td>
    </tr>
```

```
    {% empty %}
    <tr>
      No orders yet
    </tr>
  {% endfor %}
</table>
#...
```

In this code, we check for any existing order records associated with the current user. If we don't find any, we render *No records yet* element. Otherwise, we iterate through the orders and display order information:



Order information will include the order number, date of creation, total cost of the order, link to the order's invoice, and status. We will focus on order detail and order invoice links now so that the user can access additional order information through these links.

Let's start with the order invoice page, as we already implemented this page in the admin site. Open the *views.py* file of the *orders* application and create a copy of the *invoice_pdf* view (without the decorator). Name it *customer_invoice_pdf* :

finesauces/orders/views.py

```
#...
def customer_invoice_pdf(request, order_id):
    order = get_object_or_404(Order, id=order_id)

    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = f'filename=order_{order.id}.pdf'

    # generate pdf
    html = render_to_string('pdf.html', {'order': order})
    stylesheets=[weasyprint.CSS(settings.STATIC_ROOT + 'css/pdf.css')]
     weasyprint.HTML(string=html).write_pdf(response,stylesheets=stylesheets)

    return response
    #...
```

Now we need to map this view to a new URL. Open the *urls.py* file of the *orders* application and add in the following pattern:

finesauces/orders/urls.py

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

app_name = 'orders'

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

Add the *customer_invoice_pdf* URL to the *profile.html* template. Find the following code:

finesauces/accounts/templates/accounts/profile.html

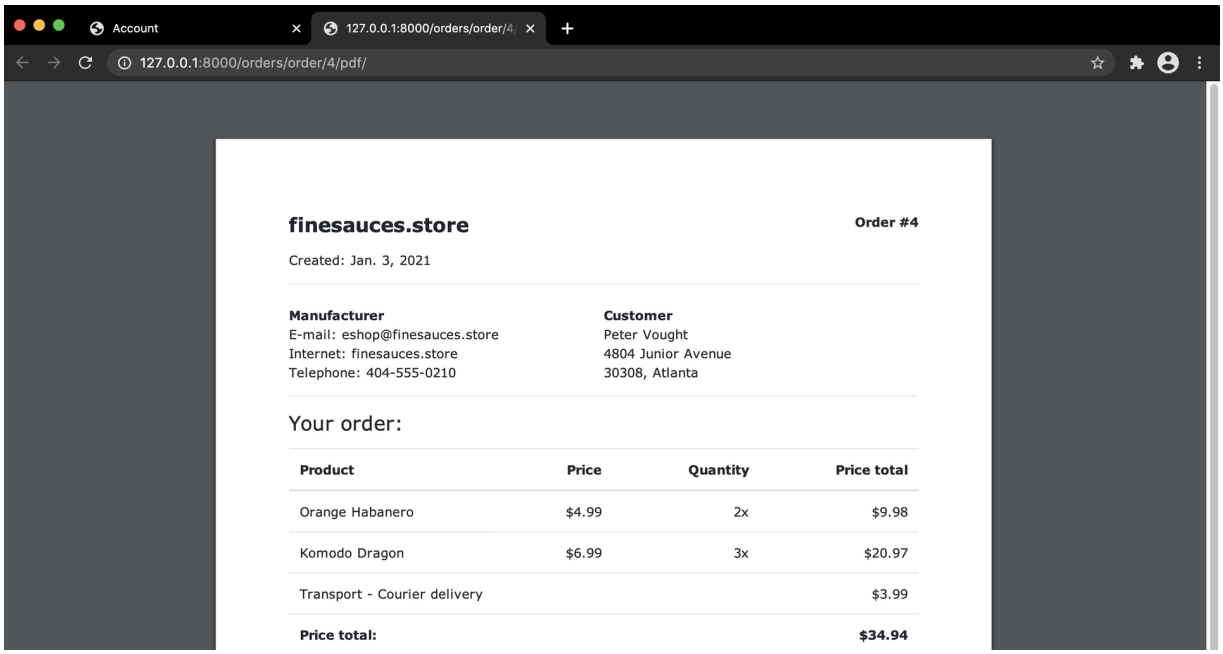
```
#...
<td>
  <a href="" class="text-decoration-none" target="_blank">
    <i class="far fa-file-pdf text-danger"></i>
    Invoice {{ order.id }}
  </a>
</td>
#...
```

Modify it to include our new URL:

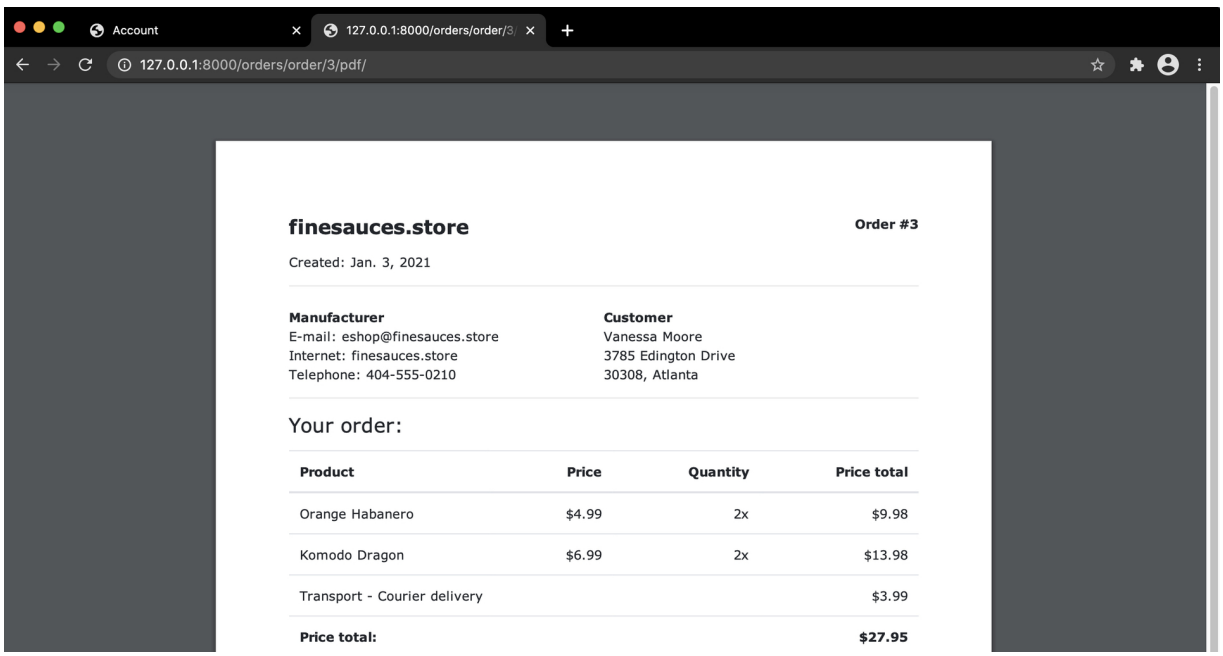
finesauces/accounts/templates/accounts/profile.html

```
#...
<td>
  <a href="{% url 'orders:customer_invoice_pdf' order.id %} "
    class="text-decoration-none" target="_blank">
    <i class="far fa-file-pdf text-danger"></i>
    Invoice {{ order.id }}
  </a>
</td>
#...
```

Let's try it out. Visit the profile section and click on the *Invoice* URL we just added for each order. You will be redirected to a new page with a rendered Invoice PDF file for the given order:



There is, however, a major underlying security issue. Try to modify the order number in the URL path to a different number. I will change it to number 3. Another user created this order, but we are still able to see the invoice!



We can fix this by implementing a simple check in the *customer_invoice_pdf* view. Open the *views.py* file in the *orders* application and modify this view to look like this:

finesauces/orders/views.py

```
from django.shortcuts import render, get_object_or_404, redirect
#...
def customer_invoice_pdf(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    if request.user == order.user:
        response = HttpResponse(content_type='application/pdf')
        response['Content-Disposition'] = f'filename=order_{order.id}.pdf'

        # generate pdf
        html = render_to_string('pdf.html', {'order': order})
        stylesheets=[weasyprint.CSS(settings.STATIC_ROOT + 'css/pdf.css')]
        weasyprint.HTML(string=html).write_pdf(response,stylesheets=stylesheets)

        return response

    return redirect('profile')
    #...
```

We implemented a simple check to evaluate if the user requesting this invoice view for a given order is actually the customer that placed that order. If not, we redirect the user to the profile page.

Check that we implemented is perfectly fine for this task. The issue might arise when we need to perform multiple checks within a single view. It might overcomplicate things and impact the maintainability of the codebase. It would be much better if we could create our custom decorator, like, for example, *@staff_member_required,* which limits access to the staff users, but in our case for the customers eligible to see the invoice for their order. Luckily, it is pretty easy to do so.

Let's create a *decorators.py* file inside the *finesauces_project* directory and add in the following decorator code:

finesauces/finesauces_project/decorators.py

```
from orders.models import Order
from django.shortcuts import redirect

def user_created_order(view_func):
    def wrap(request, *args, **kwargs):

        order_id = kwargs["order_id"]

        try:
            order = Order.objects.get(id=order_id, user=request.user)
        except Order.DoesNotExist:
            return redirect('profile')

        return view_func(request, *args, **kwargs)

    return wrap
```

Inside our *user_created_order* decorator, we define the *wrap* function, which retrieves the *order_id* parameter from our *customer_invoice_pdf* view. We then search for the order associated with the provided id and user reference. If we manage to find such an order, we then proceed to our view. Otherwise, we redirect the user to the *profile* page.

Let's apply this decorator to the *customer_invoice_pdf* view. Open *views.py* file inside *orders* application and add the following import:

finesauces/orders/views.py

```
from finesauces_project.decorators import user_created_order
```

Now remove *request.user == user.user* check and apply *user_created_order* decorator to the view:

finesauces/orders/views.py

```
@user_created_order
def customer_invoice_pdf(request, order_id):
    order = get_object_or_404(Order, id=order_id)

    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = f'filename=order_{order.id}.pdf'

    # generate pdf
    html = render_to_string('pdf.html', {'order': order})
    stylesheets=[weasyprint.CSS(settings.STATIC_ROOT + 'css/pdf.css')]
    weasyprint.HTML(string=html).write_pdf(response,stylesheets=stylesheets)

    return response
```

If we try to access Order #3 again, we are redirected back to our profile!

However, we ended up with two identical views, *invoice_pdf* and *customer_invoice_pdf* . They only differ in a decorator that is being applied. It would be great to get away using only one view and decorate it differently based on the URL user is trying to access. We want to use *@staff_member_required* when a staff member tries to access invoices inside the admin area and *@user_created_order* decorator when the user wants to check available invoices in the profile section.

Lucky for us, the Django team had this case scenario in mind, and we can resolve this in our *urls.py* file inside the *orders* application. Before venturing there, let's clean the *views.py* file a little. Remove *decorators* imports:

finesauces/orders/views.py

```
from django.contrib.admin.views.decorators import staff_member_required
from finesauces_project.decorators import user_created_order
```

Then remove the whole *customer_invoice_pdf* view, including the decorator, and also remove the decorator from the *invoice_pdf* view.

Now open the *urls.py* file and modify it to look as follows:

finesauces/orders/urls.py

```
from django.urls import path
from . import views
from django.contrib.admin.views.decorators import staff_member_required
from finesauces_project.decorators import user_created_order

app_name = 'orders'

urlpatterns = [
    path('create/', views.order_create, name='order_create'),
    path('admin/order/<int:order_id>/pdf/',
        staff_member_required( views.invoice_pdf) , name='invoice_pdf'),
    path('order/<int:order_id>/pdf/',
        user_created_order( views.invoice_pdf) , name='customer_invoice_pdf'),
]
```

We managed to utilize the same invoice view for admin and profile pages. We decorate it based on the URL pattern, which is being requested.

Lastly, let's add a simple order detail page containing similar data as the invoice, but it will not be presented as a PDF file. Open *views.py* file inside *orders* application and create the following view:

finesauces/orders/views.py

```
#...
def order_detail(request, order_id):
    order = Order.objects.get(pk=order_id)

     return render(
        request,
        'order_detail.html',
        {'order': order}
    )
#...
```

Note that we don't need to use the *get_object_or_404* function here since our decorator *@user_created_order* will handle access and order validation for us. If we pass the decorator check and get to the view, we can be sure that the requested order exists.

Now add the following decorated URL pattern to the *urls.py* file inside the *orders* application:

finesauces/orders/urls.py

```
#...
path('order/<int:order_id>/',
    user_created_order(views.order_detail), name='order_detail'),
```

Our view and URL mapping are in place. Now we need to create template *detail.html* in *templates* folder inside *orders* application. Add the following code to it:

finesauces/orders/templates/order_detail.html

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

{% block title %}Order #{{ order.id }}{% endblock %}

{% block content %}
<div class="card shadow-custom border-0 col-lg-7 mx-auto mb-3">
  <div class="mt-2">
    <span class="h2 font-weight-bold muted order_logo">
      finesauces.store
    </span>
    <span class="float-right font-weight-bold order_number">
      Order #{{ order.id }}
    </span>
  </div>
  <div class="mt-3">
     Created: {{ order.created|date }}
  </div>
  <hr>
  <div class="row justify-content-center">
    <div class="col-lg-6">
      <div class="font-weight-bold">
        Manufacturer
      </div>
      E-mail: eshop@finesauces.store<br>
      Internet: finesauces.store<br>
      Telephone: 404-555-0210<br>
    </div>
```

```
    <div class="col-lg-6">
      <div class="font-weight-bold">
        Buyer
      </div>
      {{ order.first_name }} {{ order.last_name }}<br>
      {{ order.address }}<br>
      {{ order.postal_code }}, {{ order.city }}
    </div>
  </div>
  <hr>
  <h3>Your order:</h3>
  <table class="table mt-3">
    <thead class="thead-detail">
      <tr>
        <th>Product</th>
        <th class="text-right">Price</th>
        <th class="text-right">Quantity</th>
        <th class="text-right">Price total</th>
      </tr>
    </thead>
    <tbody class="tbody-detail">
      {% for item in order.items.all %}
      <tr>
        <td class="order_product">
          {{ item.product.name }}
        </td>
        <td class="num text-right">
          ${{ item.price }}
        </td>
        <td class="num text-right">
           {{ item.quantity }}x
        </td>
        <td class="num text-right">
          ${{ item.get_cost }}
        </td>
      </tr>
      {% endfor %}
      <tr>
        <td colspan=3>
          Transport - {{ order.transport }}
        </td>
        <td class="num text-right">
          ${{ order.transport_cost }}
        </td>
      </tr>
      <tr class="total font-weight-bold">
        <td colspan="3">Price total:</td>
        <td class="num text-right">
          ${{ order.get_total_cost|floatformat:2 }}
        </td>
      </tr>
    </tbody>
  </table>
</div>
{% endblock %}
```

As a last step, we have to add the '*order_detail*' URL into profile template for each order. Let's modify *Order* model to include canonical URL by implementing *get_absolute_url()* method.

Open the *models.py* file inside *orders* application. Add the following import to the top of the file along with method to the *Order* model:

finesauces/orders/models.py

```
from django.urls import reverse
#...
class Order(models.Model):
    #...
    def get_absolute_url(self):
        return reverse(
            'orders:order_detail',
             args=[self.id]
        )
    #...
```

Open *profile.html* template inside the *accounts* directory and find the following section:

finesauces/accounts/templates/accounts/profile.html

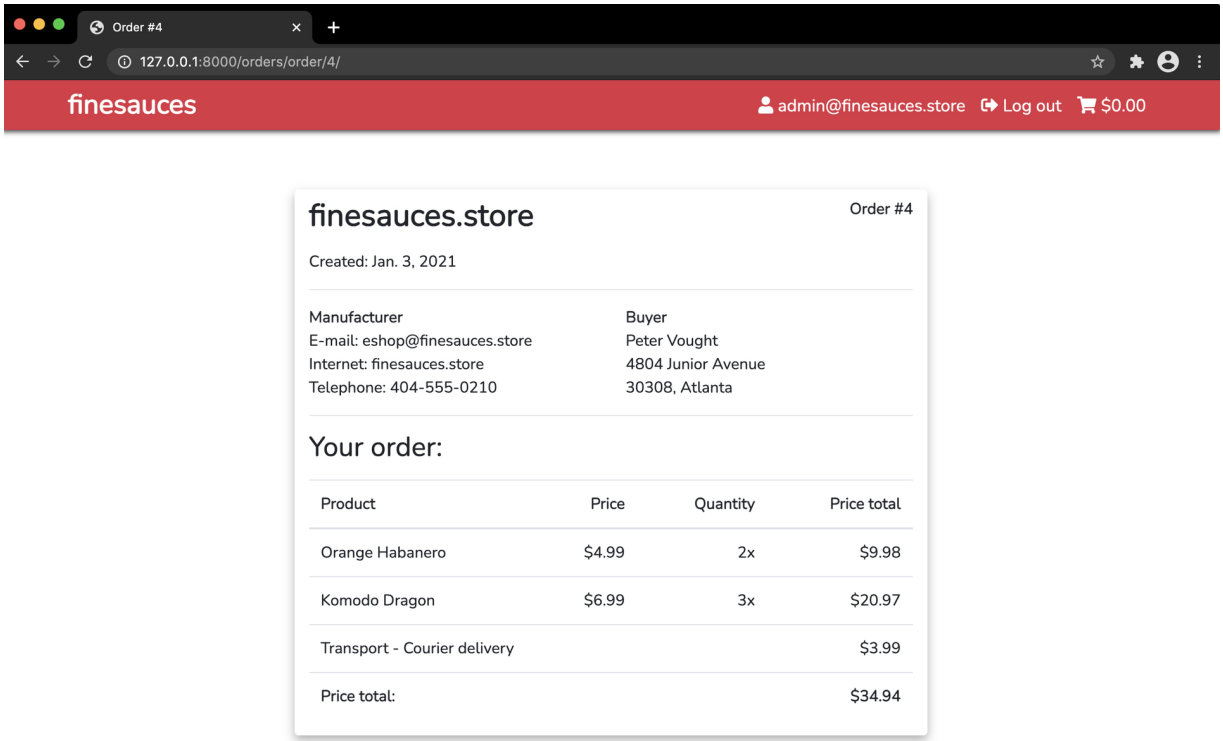
```
#...
<td>
  <a href="" class="text-decoration-none">
    {{ order.id }}
  </a>
</td>
#...
```

Update it the following way to include order detail URL:

finesauces/accounts/templates/accounts/profile.html

```
#...
<td>
  <a href="{{ order.get_absolute_url }} " class="text-decoration-none">
    {{ order.id }}
  </a>
</td>
#...
```

If we now click on the order number link in the order list section, we will be redirected to the order detail page:



Great! Our project is now officially complete. To wrap this section and prepare our e-commerce site for deployment, let's create a list of all Python libraries that this project relies on. Use the following *freeze* command to create a *requirements.txt* file, which we will use to install our Python dependencies in a production environment:

terminal

```
(env) ~/finesauces$ pip freeze > requirements.txt
```

Don't forget to push your progress to the remote repository.

8 Deploying to DigitalOcean

The deployment process might get tricky, and opportunities for error are easy to find. The following chapter serves as a step-by-step deployment guide and is based on DigitalOcean [58] documents *Initial Server Setup with Ubuntu 20.04* [59] and *How To Set Up Django with Postgres, Nginx, and Gunicorn on Ubuntu 20.04* [60] .

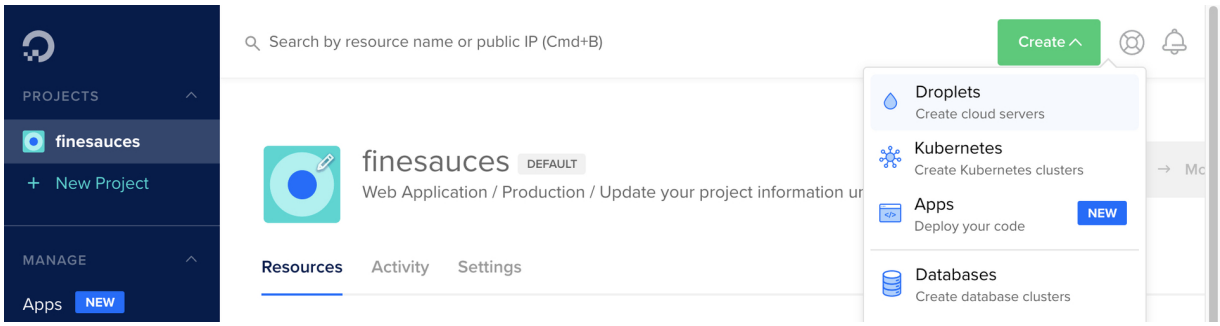
We will start by setting up our Virtual Private Server along with the PostgreSQL database and virtual environment. We will configure the *Gunicorn* [61] application server to interface with our applications and set up *Nginx* [62] to reverse proxy to Gunicorn, giving us access to its security and performance features to serve our apps. As a final step, we will go through custom domain setup and adding an SSL certificate to our website.

8.1 VPS access and security

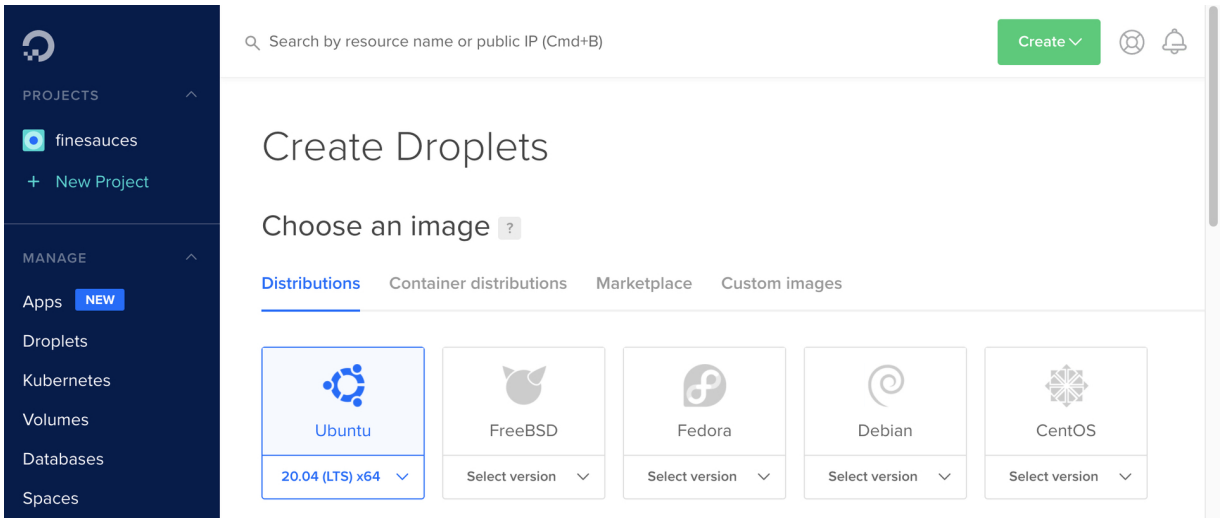
We will use the *DigitalOcean* cloud hosting provider to house our project. If you haven't created your account yet, you can use this link [63] to register and get free $100 credit to start.

8.1.1 Creating Droplet

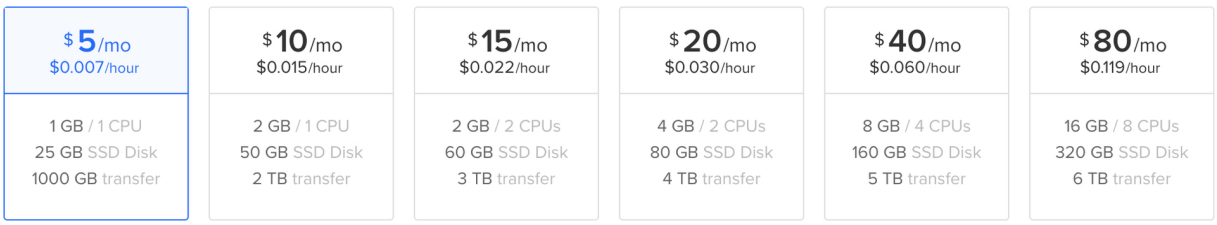
Once registered and signed in, select *Droplets* from the dropdown menu:



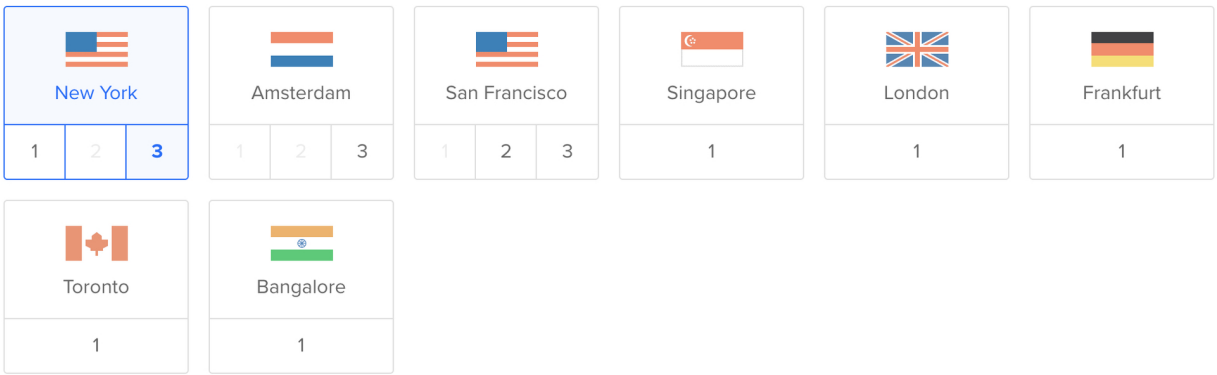
We are then redirected to the Droplet creation page. Droplets are basically instances of various Linux distributions.



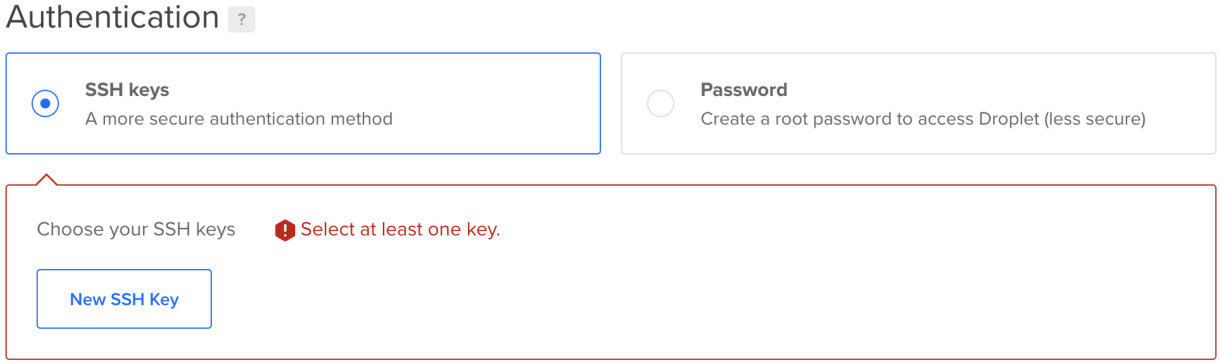
Select *Ubuntu 20.04 (LTS)* version and *Basic* plan. We will opt for a $5/month plan, which is more than enough for our needs.



We won't be adding any additional storage, so ignore the *Add block storage* option. For the *datacenter region* , select the one closest to you.



As for *Authentication* , we will be using SSH keys to set up and secure our connection to the server.



8.1.2 Creating SSH key

We don't have any SSH keys available as of now. Let's go ahead and create one. Open your terminal and type the following command:

```
terminal
~$ ssh-keygen
```

By default, we will be offered to create *id_rsa* and *id_rsa.pub* files. I prefer to keep my ssh keys separated, so I will specify a different name for these files:

```
terminal
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/peter/.ssh/id_rsa): .ssh/id_rsa_do
```

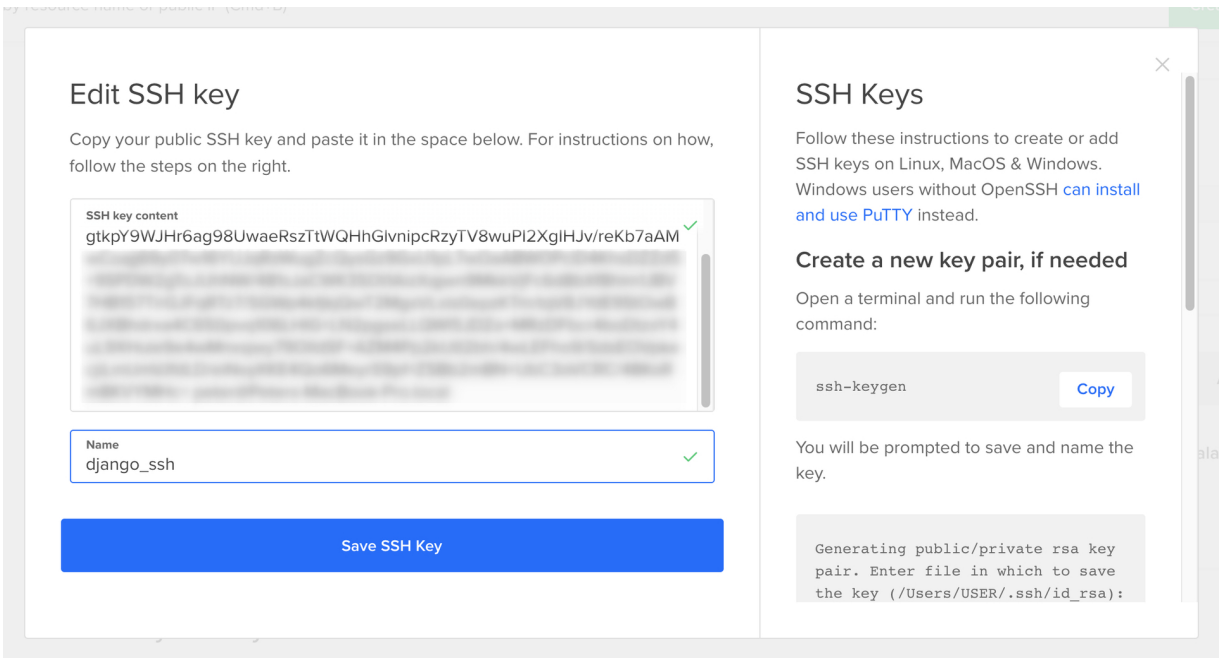
And following private and public ssh keys are generated for me:

```
~/.ssh/id_rsa_do
~/.ssh/id_rsa_do.pub
```

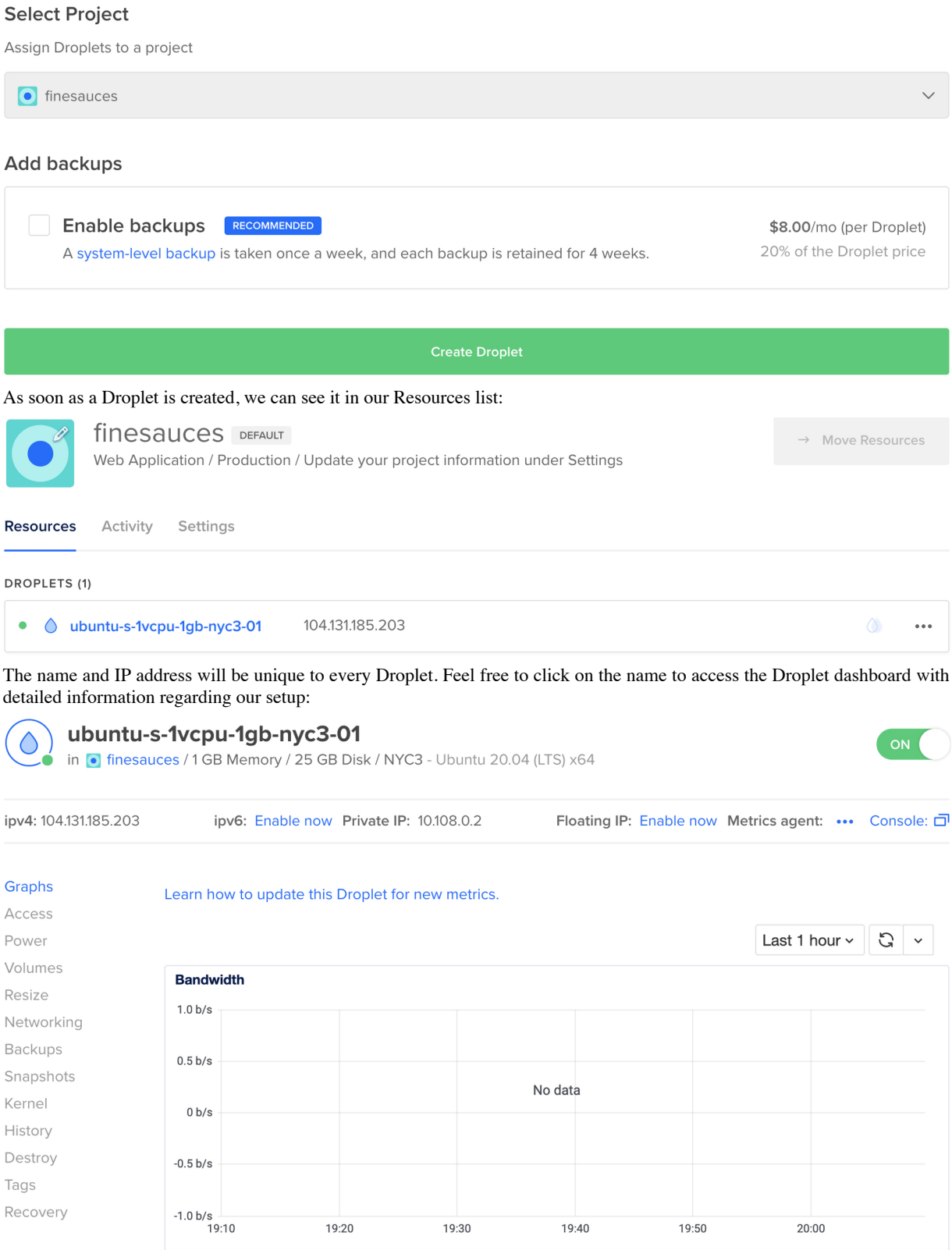
Let's copy the contents of the *id_rsa.pub* key by using the following command:

```
terminal
~$ cat ~/.ssh/id_rsa_do.pub
```

This command will reveal the content of our public ssh key file in the terminal. Return back to DigitalOcean, click on *New SSH Key*, and paste the content to the SSH key window. Provide some name for this key:



Once done with these steps, click on *Add SSH Key* to add a new SSH key to our account. Let's finish this setup by creating our Droplet by clicking on the *Create Droplet* button down below:



As soon as a Droplet is created, we can see it in our Resources list:

The name and IP address will be unique to every Droplet. Feel free to click on the name to access the Droplet dashboard with detailed information regarding our setup:

8.1.3 Logging into Droplet

Let's try connecting to our Droplet. Open terminal and type the following command (replace *104.131.185.203* with your IP address):

```
terminal
~$ ssh root@104.131.185.203
```

When logging in for the first time, you will most probably receive the following notification:

```
terminal
The authenticity of host '104.131.185.203 (104.131.185.203)' can't be established.
ECDSA key fingerprint is SHA256:B8ePWNEy7jjmhamtQkHi1w6HsumydXKxftmrWD4ufj8.
Are you sure you want to continue connecting (yes/no/[fingerprint])? Yes
```

Confirm with *yes*.

You might also receive the following message:

terminal
```
root@ 104.131.185.203: Permission denied(publickey)
```

In this case, run the following command to add the key to ssh-agent:

terminal
```
~$ ssh-add ~/.ssh/id_rsa_do
```

and try to connect again:

terminal
```
~$ ssh root@104.131.185.203
```

Upon successfully logging into our server, we are greeted by the following report:

Ubuntu 20.04.5 LTS terminal
```
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-51-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sun Jan  3 19:15:11 UTC 2021

  System load:  0.0               Users logged in:       0
  Usage of /:   5.1% of 24.06GB   IPv4 address for eth0: 104.131.185.203
  Memory usage: 19%               IPv4 address for eth0: 10.17.0.5
  Swap usage:   0%                IPv4 address for eth1: 10.108.0.2
  Processes:    100

1 update can be installed immediately.
0 of these updates are security updates.
To see these additional updates run: apt list --upgradable


The list of available updates is more than a week old.
To check for new updates run: sudo apt update


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

8.1.4 Creating a new user and updating security settings

We don't want to use the root user for handling our project on the server for security reasons. The root user is the administrative user in a Linux environment that has very broad privileges. Because of the heightened privileges of the *root* account, we are discouraged from using it regularly. This is because part of the power inherent with the root account is the ability to make very destructive changes, even by accident.

Let's go ahead and create a new user account now. Feel free to use any name you like. I will go with *finesaucesadmin* . Use the following command to create the user:

Ubuntu 20.04.5 LTS terminal
```
# adduser finesaucesadmin
```

Follow instructions in the terminal. Make sure you provide some secure password. You can go with the default, blank values for contact information fields:

Ubuntu 20.04.5 LTS terminal
```
Adding user `finesaucesadmin' ...
Adding new group `finesaucesadmin' (1000) ...
Adding new user `finesaucesadmin' (1000) with group `finesaucesadmin' ...
Creating home directory `/home/finesaucesadmin' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for finesaucesadmin
Enter the new value, or press ENTER for the default
   Full Name []:
   Room Number []:
   Work Phone []:
   Home Phone []:
   Other []:
Is the information correct? [Y/n] Y
```

Now, we have a new user account with regular account privileges. However, we may occasionally need to perform administrative tasks. Let's give our new user admin privileges. To add these privileges to our new user, we need to add the

user to the sudo group:

Ubuntu 20.04.5 LTS terminal
```
# usermod -aG sudo finesaucesadmin
```

In order to be able to log in via ssh as a new user, we need to set up SSH keys on the server. Navigate to our new users *home* folder:

Ubuntu 20.04.5 LTS terminal
```
# cd /home/finesaucesadmin
```

Create *.ssh* directory:

Ubuntu 20.04.5 LTS terminal
```
/home/finesaucesadmin# mkdir .ssh
```

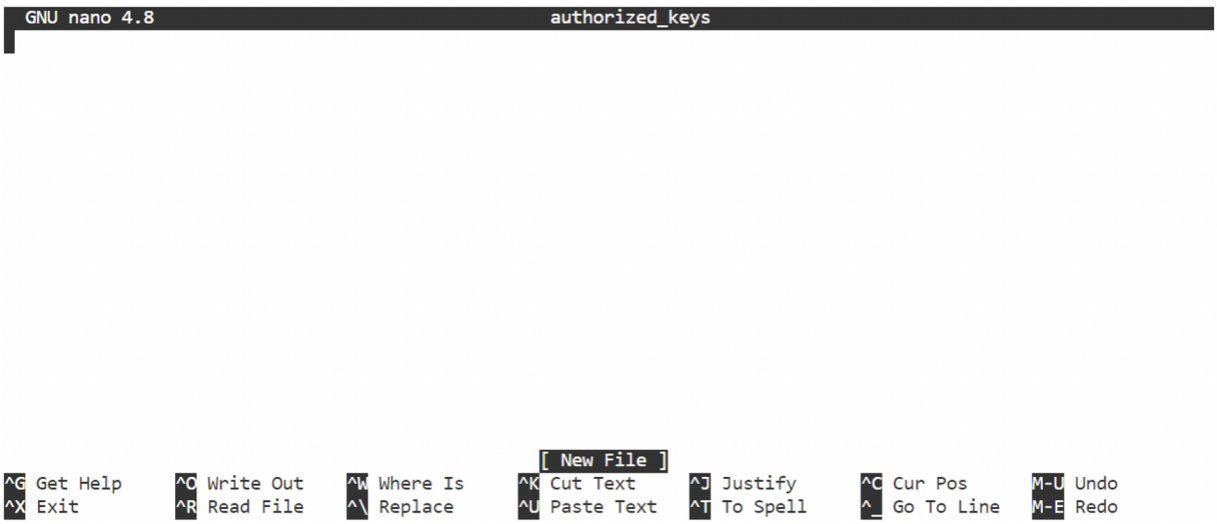
Move into this directory by using the *cd* command:

Ubuntu 20.04.5 LTS terminal
```
/home/finesaucesadmin# cd .ssh
```

Once inside *.ssh* folder, create *authorized_keys* file:

Ubuntu 20.04.5 LTS terminal
```
/home/finesaucesadmin/.ssh# nano authorized_keys
```

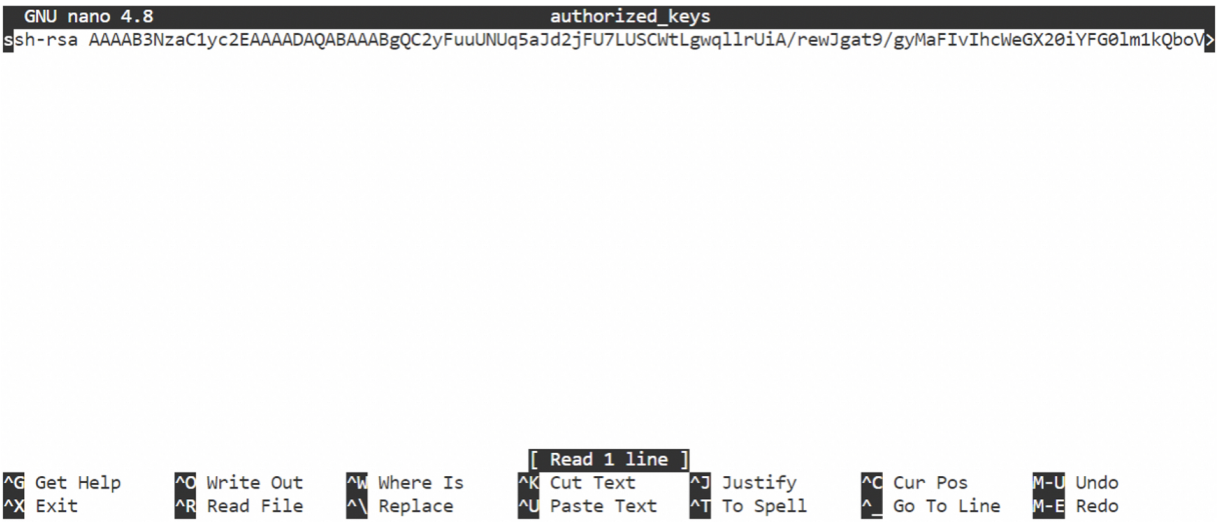
Your terminal windows will turn into a simple text editor:



Here we need to paste our SSH key. We can either generate a new one or use one we already have on our local machine. Let's use that one. Open a new terminal window and use the following command to copy the contents of the *id_rsa_do.pub* file:

terminal
```
~$ cat ~/.ssh/id_rsa_do.pub
```

Paste it to *authorized_keys* file opened in a terminal window.



Press *CONTROL-X* to save the file. Confirm changes by pressing *Y* . Then press *ENTER* to confirm *file name: authorized_keys* .

Now we should be able to log in as a new user. First disconnect from the server:

Ubuntu 20.04.5 LTS terminal
```
/home/finesaucesadmin/.ssh# exit
```

And log in as a new user:

terminal
```
~$ ssh finesaucesadmin@104.131.185.203
```

After successfully logging in as a new user, we need to disable root login. Use the following command to open the SSHD config file:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo nano /etc/ssh/sshd_config
```

Find the *PermitRootLogin* and *PasswordAuthentication* attributes and set them to no. Save the file, and let's reload the SSHD service for changes to take effect:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo systemctl reload sshd
```

Ubuntu 20.04 servers can use the UFW firewall to make sure only connections to certain services are allowed. We can set up a basic firewall very easily using this application. Applications can register their profiles with UFW upon installation. These profiles allow UFW to manage these applications by name. OpenSSH, the service allowing us to connect to our server now, has a profile registered with UFW. We can verify this by using the following command:

Ubuntu 20.04.5 LTS terminal.
```
~$ sudo ufw app list
```

You should see similar output:

Ubuntu 20.04.5 LTS terminal
```
Available applications:
  OpenSSH
```

We need to make sure that the firewall allows SSH connections so that we can log back in next time:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo ufw allow OpenSSH
```

You should see the following message:

Ubuntu 20.04.5 LTS terminal
```
Rules updated
Rules updated (v6)
```

Now we need to enable firewall:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo ufw enable
```

When asked about proceeding further, select y:

Ubuntu 20.04.5 LTS terminal
```
Command may disrupt existing ssh connections. Proceed with operation (y|n)? y
Firewall is active and enabled on system startup
```

Let's check the status of our firewall to confirm everything is working:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo ufw status
```

The following confirmation should appear in your terminal:

Ubuntu 20.04.5 LTS terminal
```
Status: active

To                         Action      From
--                         ------      ----
OpenSSH                    ALLOW       Anywhere
OpenSSH (v6)               ALLOW       Anywhere (v6)
```

As the firewall is currently blocking all connections except for SSH, if you install and configure additional services, you will need to adjust the firewall settings to allow traffic in.

8.2 Installing software

We will start by updating our server packages. Run the following commands to get up to date:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo apt update
~$ sudo apt upgrade
```

You will be asked about installing new packages. Select Y to confirm:

Ubuntu 20.04.5 LTS terminal
```
After this operation, 175 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

Let's install Python3, Postgres, NGINX and rabbitmq-server now. Use the following command:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo apt install python3-pip python3-dev libpq-dev postgresql
postgresql-contrib nginx curl rabbitmq-server
```

Select Y to confirm installation:

Ubuntu 20.04.5 LTS terminal
```
After this operation, 608 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

After installation finishes, we can continue by installing *WeasyPrint* dependencies:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo apt-get install build-essential python3-dev python3-pip
python3-setuptools python3-wheel python3-cffi libcairo2 libpango-1.0-0
libpangocairo-1.0-0 libgdk-pixbuf2.0-0 libffi-dev shared-mime-info
```

Confirm the installation to continue:

Ubuntu 20.04.5 LTS terminal
```
After this operation, 7695 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

8.2.1 Database setup

Let's set up our PostgreSQL database. Log in to the database session:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo -u postgres psql
```

and create *finesauces* database:

postgres terminal
```
postgres=# CREATE DATABASE finesauces;
```

Create a user for the database (I will use the same credentials here as on my local machine during development):

postgres terminal
```
postgres=# CREATE USER finesaucesadmin WITH PASSWORD '********';
```

As we already did at the beginning of our project during local development setup, set default encoding, transaction isolation scheme, and timezone (Recommended from Django team):

postgres terminal
```
postgres=# ALTER ROLE finesaucesadmin SET client_encoding TO 'utf8';
postgres=# ALTER ROLE finesaucesadmin SET default_transaction_isolation TO
'read committed';
postgres=# ALTER ROLE finesaucesadmin SET timezone TO 'UTC';
```

Grant *finesaucesadmin* user access to administer our database:

postgres terminal
```
postgres=# GRANT ALL PRIVILEGES ON DATABASE finesauces TO finesaucesadmin;
```

We can quit the PostgreSQL session now.

postgres terminal
```
postgres=# \q
```

8.3 Virtual environment

Before cloning up our project from the remote repository, we need to set up our virtual environment. To do so, we first have to install the python3 *venv* package. Install the package by typing:

Ubuntu 20.04.5 LTS terminal
```
~$ sudo apt-get install python3-venv
```

After installing the *venv* package, we can proceed by creating and moving into our *django_projects* project directory, where our current and future projects will reside:

Ubuntu 20.04.5 LTS terminal
```
~$ mkdir django_projects
~$ cd django_projects
```

When inside our new project directory, clone remote repository (replace the link with your repository URL):

Ubuntu 20.04.5 LTS terminal
```
~/django_projects$ git clone https://github.com/Peter-Vought/finesauces.git
```

Once our project is copied into the *django_projects* directory, move inside that project:

Ubuntu 20.04.5 LTS terminal
```
~/django_projects$ cd finesauces/
```

and create a virtual environment:

Ubuntu 20.04.5 LTS terminal
```
~/django_projects/finesauces$ python3 -m venv env
```

Activate the environment:

Ubuntu 20.04.5 LTS terminal
```
~/django_projects/finesauces$ source env/bin/activate
```

Now we can go ahead and install our Python dependencies listed in the *requirements.txt* file:

Ubuntu 20.04.5 LTS terminal
```
(env)~/django_projects/finesauces$ pip install -r requirements.txt
```

8.3.1 Settings and migrations

Let's create a *local_settings.py* file to store project sensitive information. Move into *finesauces_project* folder:

Ubuntu 20.04.5 LTS terminal
```
(env)~/django_projects/finesauces$ cd finesauces_project/
```

Create *local_settings.py* file by using the following command:

Ubuntu 20.04.5 LTS terminal
```
(env)~/django_projects/finesauces/finesauces_project$ sudo nano local_settings.py
```

Paste in the contents from your local machine *local_settings.py* file and update *DEBUG* field to *False* and add your *Droplet* IP address to the *ALLOWED_HOSTS* list:

```
#...
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = False

ALLOWED_HOSTS = ['104.131.185.203' ]
#...
```

Now we need to move back to the *finesauces* directory:

Ubuntu 20.04.5 LTS terminal
```
(env)~/django_projects/finesauces/finesauces_project$ cd ..
```

and run initial migrations for our project:

Ubuntu 20.04.5 LTS terminal
```
(env)~/django_projects/finesauces$ python manage.py migrate
```

If everything was set up correctly, you should see the following output in the terminal:

Ubuntu 20.04.5 LTS terminal
```
Operations to perform:
  Apply all migrations: account, admin, auth, contenttypes,
listings, orders, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying account.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 listings.0001_initial... OK
  Applying listings.0002_auto_20201019_1104... OK
  Applying orders.0001_initial... OK
  Applying orders.0002_order_user... OK
  Applying sessions.0001_initial... OK
```

Now let's create our superuser with the *createsuperuser* command:

Ubuntu 20.04.5 LTS terminal
```
(env)~/django_projects/finesauces$ python manage.py createsuperuser
```

Prepare static files to be served by the server:

Ubuntu 20.04.5 LTS terminal
```
(env)~/django_projects/finesauces$ python manage.py collectstatic
```

8.4 Gunicorn setup

Let's install Gunicorn by using the *pip* package manager:

Ubuntu 20.04.5 LTS terminal
```
(env)~/django_projects/finesauces$ pip install gunicorn
```

After a successful installation, we can deactivate the virtual environment:

Ubuntu 20.04.5 LTS terminal
```
(env)~/django_projects/finesauces$ deactivate
```

To implement a way to start and stop our application server, we will create *system service* and *socket* files. The Gunicorn socket will be created at boot and will listen for connections. When a connection occurs, the system will automatically start the Gunicorn process to handle the connection.

Open systemd socket file for Gunicorn called *gunicorn.socket* :

Ubuntu 20.04.5 LTS terminal
```
~/django_projects/finesauces$ sudo nano /etc/systemd/system/gunicorn.socket
```

Inside, we will create a *[Unit]* section to describe the socket, a *[Socket]* section to define the socket location, and an *[Install]* section to make sure the socket is created at the right time. Paste in the following code and save the file once done:

/etc/systemd/system/gunicorn.socket
```
[Unit]
Description=gunicorn socket

[Socket]
ListenStream=/run/gunicorn.sock

[Install]
WantedBy=sockets.target
```

Now create and open *gunicorn.service* file:

Ubuntu 20.04.5 LTS terminal
```
~/django_projects/finesauces$ sudo nano /etc/systemd/system/gunicorn.service
```

Copy this code, paste it in and save the file:

/etc/systemd/system/gunicorn.service
```
[Unit]
Description=gunicorn daemon
Requires=gunicorn.socket
After=network.target

[Service]
User=finesaucesadmin
Group=www-data
WorkingDirectory=/home/finesaucesadmin/django_projects/finesauces
ExecStart=/home/finesaucesadmin/django_projects/finesauces/env/bin/gunicorn \
          --access-logfile - \
          --workers 3 \
          --bind unix:/run/gunicorn.sock \
          finesauces_project.wsgi:application

[Install]
WantedBy=multi-user.target
```

The *[Unit]* section is used to specify metadata and dependencies. It contains a description of our service and tells the *init* system to start this after the networking target has been reached. Because our service relies on the socket from the socket file, we need to include a *Requires* directive to indicate that relationship:

In the *[Service]* part, we specify the user and group that we want the process to run under. We give our *finesaucesadmin* ownership of the process since it owns all of the relevant files. We'll give group ownership to the *www-data* group so that Nginx can communicate easily with Gunicorn. We'll then map out the working directory and specify the command to use to start the service. In this case, we'll have to specify the full path to the Gunicorn executable, which is installed within our virtual environment. We will bind the process to the Unix socket we created within the /run directory so that the process can communicate with Nginx. We log all data to standard output so that the journald process can collect the Gunicorn logs. We can also specify any optional Gunicorn tweaks here. For example, we specified 3 worker processes in this case.

[Install] section will tell system what to link this service to if we enable it to start at boot.

We can now start and enable Gunicorn socket:

Ubuntu 20.04.5 LTS terminal
```
~/django_projects/finesauces$ sudo systemctl start gunicorn.socket
~/django_projects/finesauces$ sudo systemctl enable gunicorn.socket
```

After running the *enable* command, you should see the similar output:

Ubuntu 20.04.5 LTS terminal
```
Created symlink /etc/systemd/system/sockets.target.wants/gunicorn.socket →
/etc/systemd/system/gunicorn.socket
```

Check the status of *gunicorn* to confirm whether it was able to start:

Ubuntu 20.04.5 LTS terminal
```
~/django_projects/finesauces$ sudo systemctl status gunicorn.socket
```

If everything was set up properly, you should see similar output:

Ubuntu 20.04.5 LTS terminal
```
● gunicorn.socket - gunicorn socket
     Loaded: loaded (/etc/systemd/system/gunicorn.socket; enabled;
vendor preset: enabled)
     Active: active (listening) since Sun 2021-01-03 20:11:09 UTC; 22s ago
   Triggers: ● gunicorn.service
     Listen: /run/gunicorn.sock (Stream)
      Tasks: 0 (limit: 1137)
     Memory: 0B
     CGroup: /system.slice/gunicorn.socket

Jan 03 20:11:09 ubuntu-s-1vcpu-1gb-nyc3-01 systemd[1]:
Listening on gunicorn socket.
```

8.5 NGINX setup

Now that Gunicorn is set up, we need to configure Nginx to pass traffic to the process. We will start by creating and opening a new server block in Nginx's sites-available directory:

Ubuntu 20.04.5 LTS terminal
```
~/django_projects/finesauces$ sudo nano /etc/nginx/sites-available/finesauces
```

Paste in the following code. Make sure you provide your Droplet IP address in the *server_name* attribute:

/etc/nginx/sites-available/finesauces

```
server {
    listen 80;
    server_name 104.131.185.203;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        root /home/finesaucesadmin/django_projects/finesauces;
    }

    location /media/ {
        root /home/finesaucesadmin/django_projects/finesauces;
    }

    location / {
        include proxy_params;
        proxy_pass http://unix:/run/gunicorn.sock;
    }
}
```

We specify that this block should listen on port 80, and it should respond to our Droplet's IP address. Next, we will tell Nginx to ignore any problems with finding a favicon. We will also tell it where to find the static assets that we collected in our *~/finesauces/static* directory. All of these files have a standard URI prefix of "/static", so we can create a location block to match those requests. Finally, we'll create a *location / {}* block to match all other requests. Inside of this location, we'll include the standard *proxy_params* file included with the Nginx installation, and then we will pass the traffic directly to the Gunicorn socket.

Enable this file by linking it to the *sites-enabled* dir:

Ubuntu 20.04.5 LTS terminal

```
~/django_projects/finesauces$ sudo ln -s /etc/nginx/sites-available/finesauces
/etc/nginx/sites-enabled
```

Test NGINX config:

Ubuntu 20.04.5 LTS terminal

```
~/django_projects/finesauces$ sudo nginx -t
```

You should see the following output:

Ubuntu 20.04.5 LTS terminal

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

Restart NGINX:

Ubuntu 20.04.5 LTS terminal

```
~/django_projects/finesauces$ sudo systemctl restart nginx
```

Open up our firewall to allow normal traffic on port 80:

Ubuntu 20.04.5 LTS terminal

```
~/django_projects/finesauces$ sudo ufw allow 'Nginx Full'
```

This should be the terminal output:

Ubuntu 20.04.5 LTS terminal

```
Rule added
Rule added (v6)
```

Now we can start rabbitmq-server and Celery:

Ubuntu 20.04.5 LTS terminal

```
~/django_projects/finesauces$ sudo rabbitmq-server
```

You will probably receive notification that *rabbitmq-server* is already running:

Ubuntu 20.04.5 LTS terminal

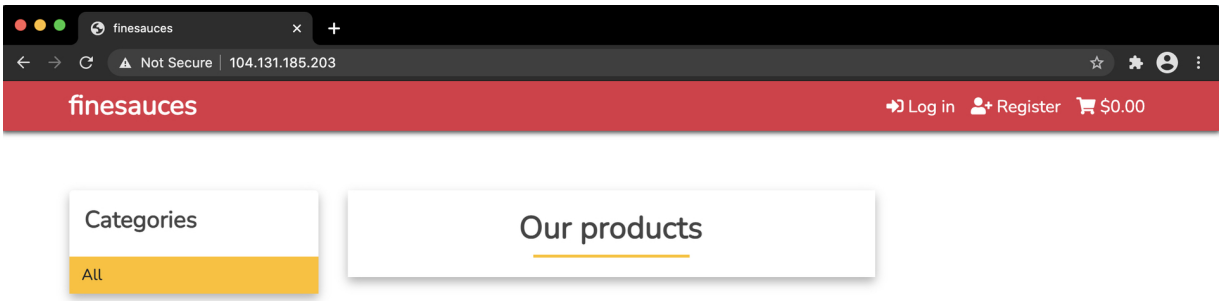
```
ERROR: node with name "rabbit" already running on "ubuntu-s-1vcpu-1gb-nyc3-01"
```

To start Celery task manager, make sure your virtual environment is active, and you are within your main project directory folder:

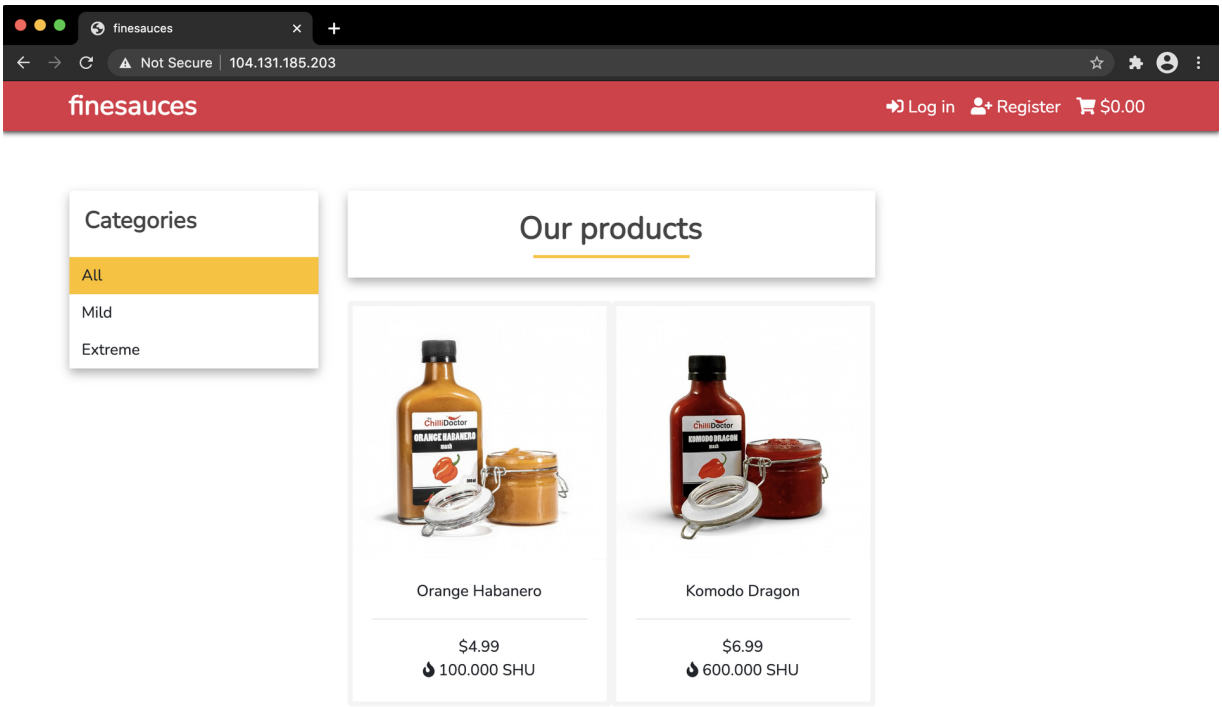
Ubuntu 20.04.5 LTS terminal

```
(env)~/django_projects/finesauces$ celery -A finesauces_project worker -l info
```

Our e-commerce project is now successfully deployed. Let's try to access our site by using Droplet IP address http://104.131.185.203 :



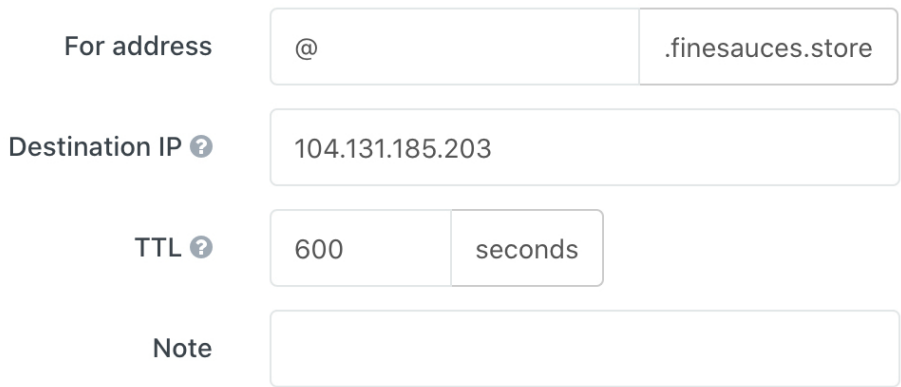
We have not added any *Category* or *Product* yet, so let's visit our Admin section at http://104.131.185.203/admin/ and add some.



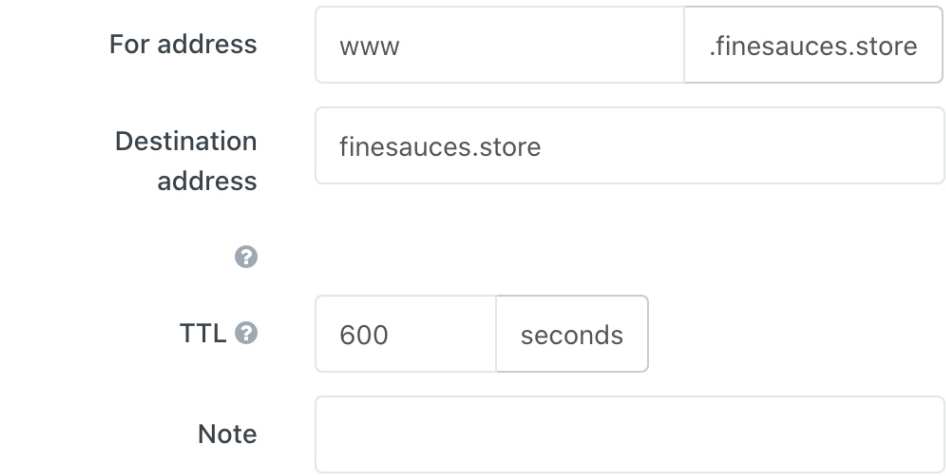
8.6 Domain setup

To obtain a custom domain, we will need to use the service of a domain registrar. There are many options to choose from. One of them would be *Namecheap* [64] . You can use whichever registrar you like. All of their interfaces will look almost identical.

Once you choose your domain registrar, log in to your account, and purchase a custom domain that you like. Inside the domain dashboard, look for *DNS* settings. There, we will need to create *A* and *CNAME records:* We will start with *A* record. For *Host* value, use @ symbol, and for *IP* address, use your Droplet's IP address:



As for *CNAME,* set *Host* value to *www* . As a target value, provide your actual domain value. In my case, that would be *finesauces.store:*



Now we need to return to *local_settings.py* and update *ALLOWED_HOSTS* to include our domain:

```
Ubuntu 20.04.5 LTS terminal
~/django_projects/finesauces/finesauces_project$ sudo nano local_settings.py
```

Add the domain in the following way:

```
ALLOWED_HOSTS = ['104.131.185.203', 'finesauces.store', 'www.finesauces.store' ]
```

We also need to update */etc/nginx/sites-available/finesauces* file to include our domain:

```
Ubuntu 20.04.5 LTS terminal
~/django_projects/finesauces$ sudo nano /etc/nginx/sites-available/finesauces
```

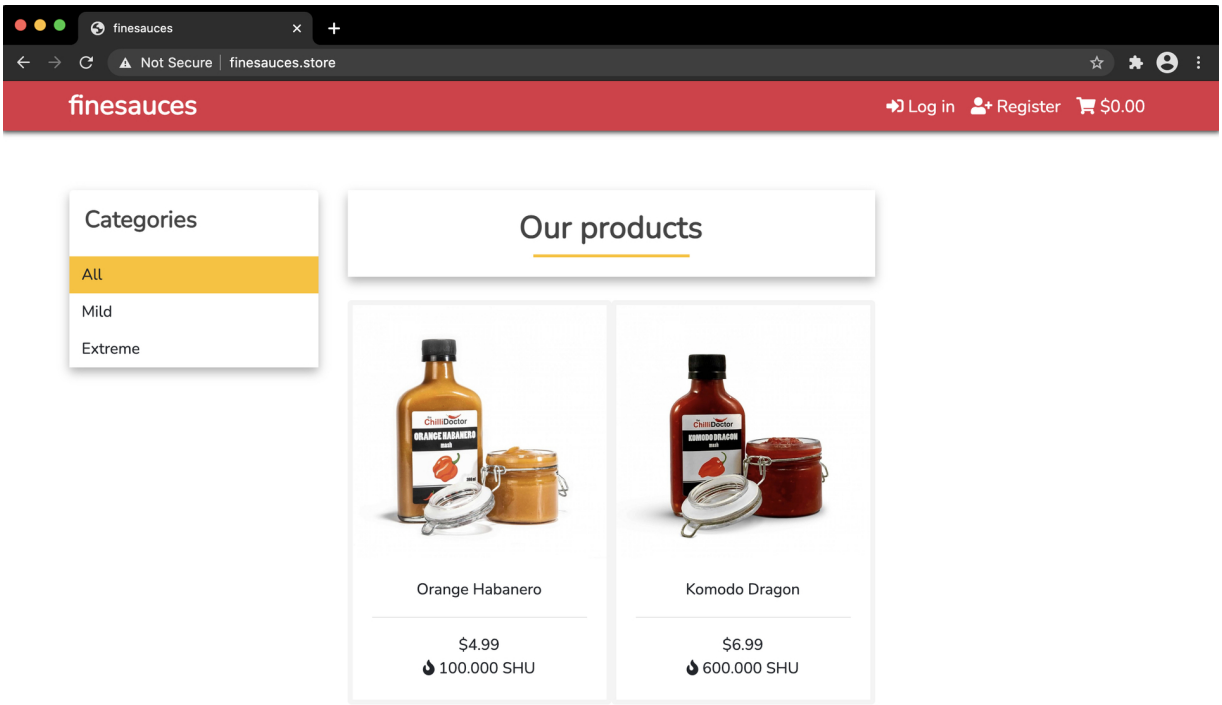
Add our domain like this next to our Droplet's IP address:

```
/etc/nginx/sites-available/finesauces
server_name 104.131.185.203 finesauces.store www.finesauces.store ;
```

Reload NGINX & Gunicorn for updates to take effect:

```
Ubuntu 20.04.5 LTS terminal
~/django_projects/finesauces$ sudo systemctl restart nginx
~/django_projects/finesauces$ sudo systemctl restart gunicorn
```

Our e-commerce site is now available at http://finesauces.store/ :



8.7 Setting up an SSL certificate

The core function of an SSL certificate is to protect server-client communication. Upon installing SSL, every bit of information is encrypted and turned into the undecipherable format. Valid SSL certification helps tremendously to establish a

trustworthy experience for visitors, which is especially important for websites accepting payments. Popular search engines also seem to favor secured sites enabling HTTPS connection.

To set up an SSL certificate and enable HTTPS, we will now install a certification tool called *Certbot* [65] , which is a free and open-source tool for using *Let's Encrypt* [66] certificates on manually-administered websites. Use the following command for installation:

```
Ubuntu 20.04.5 LTS terminal
~$ sudo snap install --classic certbot
```

After successful installation, you should see the following output:

```
Ubuntu 20.04.5 LTS terminal
certbot 1.9.0 from Certbot Project (certbot-eff✓) installed
```

Let's run the following command to obtain a certificate and have Certbot edit our Nginx configuration automatically to serve it, turning on HTTPS access in a single step:

```
Ubuntu 20.04.5 LTS terminal
~$ sudo certbot --nginx
```

Go through the terminal prompts. When asked about domain names for which you would like to activate HTTPS:

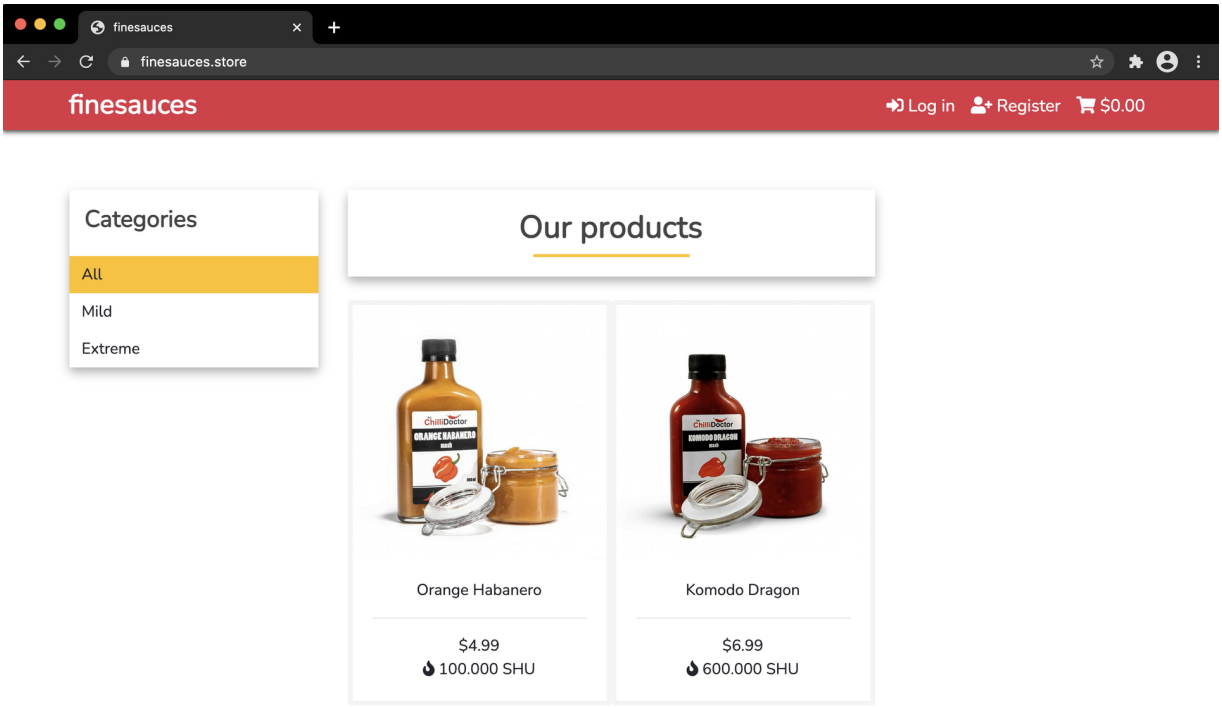
```
Ubuntu 20.04.5 LTS terminal
Which names would you like to activate HTTPS for?
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1: finesauces.store
2: www.finesauces.store
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Select the appropriate numbers separated by commas and/or spaces, or leave input
blank to select all options shown (Enter 'c' to cancel):
```

leave the input blank and press *Enter* to select all of the options. As a final step, test automatic certificate renewal by using the following command:

```
Ubuntu 20.04.5 LTS terminal
~$ sudo certbot renew --dry-run
```

Certificates will be renewed automatically before they expire. We will not need to run Certbot again, unless we change the configuration.

Our e-commerce site is now available at https://finesauces.store/ . HTTPS certificates are in place, and a lock icon is displayed in the URL bar:



The project is now complete and successfully deployed! Users are able to pick products, add them to the shopping cart, place an order, and pay for it via Stripe payment gateway. If they create an account, they will be able to review their orders in the account section. Our users are also able to change and reset their passwords if needed.

Conclusion

Congratulations on making it to the end! I sincerely hope you enjoyed the e-commerce project that we built throughout the course of this book. We have covered the essential aspects of the Django framework: URLs, views, models, templates, sessions, authentication, and wrapped it up by deploying our project into the server. You are now well suited to start building your own projects and make them become a reality.

Learning by doing is the best approach to ensuring that knowledge you obtain from any source really sticks to you. That is why I would recommend going through this project one more time, modify it, play with it, don't be afraid to eventually break and fix it. There is always a place for implementing new functionality. As food for thought and ideas for expanding this project: our products contain the attribute *"available"* indicating whether they are in stock or not. Would there be a better way to track this? Maybe use the quantity field instead? If there are more than 0 units of a product at our disposal, mark it as *"available"*. Also, allow users to only order quantity which is less or equal to what we have available in stock. And then, of course, deduct this ordered quantity from the number of units offered.

As for where to go from here, start creating your own Django projects from scratch. Make it small and manageable at first, and expand as you move forward. Build a blogging application, for example. Display articles, then enable visitors to write articles as well. Implement comments so that users can leave their feedback. Maybe user accounts? So only authenticated visitors can contribute? I will leave it up to you to decide.

The web development area is vast, and Django is only part of the puzzle. In this book, we figured out how to handle data and process it accordingly. Users, however, expect a pleasant visual presentation to go with it. That is where HTML, CSS, and optionally Javascript comes in. There is no way around it. If you plan to go the web development route, you will need to explore these technologies eventually.

As a closing note, I would love to hear back from you at *peter@vought.tech* . You are more than welcome to reach out to me and share your story and feedback. Did you enjoy this book? Did you learn something new? Is there anything that you think I missed and should be included? Was this book easy to follow? I wish you happy coding and all the best in your future endeavors. When you get stuck, don't be afraid to step away for a moment and take a deep breath.

References

[←1]
https://docs.microsoft.com/en-us/windows/wsl/about

[←2]
https://docs.microsoft.com/en-us/windows/wsl/install-win10

[←3]
https://docs.microsoft.com/en-us/windows/wsl/install-win10#step-2---check-requirements-for-running-wsl-2

[←4]
https://docs.microsoft.com/en-us/windows/wsl/troubleshooting#error-0x80370102-the-virtual-machine-could-not-be-started-because-a-required-feature-is-not-installed

[←5]
https://postgresapp.com/

[←6]
https://docs.djangoproject.com/en/3.1/ref/databases/#optimizing-postgresql-s-configuration

[←7]
https://git-scm.com/

[←8]
bitbucket.org

[←9]
github.com

[←10]
https://docs.djangoproject.com/en/3.1/topics/db/models/

[←11]
https://docs.djangoproject.com/en/3.1/topics/db/queries/

[←12]
https://docs.djangoproject.com/en/3.1/topics/db/models/#meta-options

[←13]
https://docs.djangoproject.com/en/3.1/ref/models/options/

[←14]
https://docs.djangoproject.com/en/3.1/ref/django-admin/#django-admin-sqlmigrate

[←15]
https://docs.djangoproject.com/en/3.1/ref/contrib/auth/

[←16]
https://docs.djangoproject.com/en/3.1/ref/models/options/

[←17]
https://docs.djangoproject.com/en/3.1/ref/contrib/admin#django.contrib.admin.ModelAdmin.prepopulated_fields

[←18]
http://www.cupcakeipsum.com/

[←19]
https://docs.djangoproject.com/en/3.1/topics/http/views/

[←20]
https://docs.djangoproject.com/en/3.1/topics/db/queries/

[←21]
https://docs.djangoproject.com/en/3.1/topics/templates/#the-django-template-language

[←22]
https://docs.djangoproject.com/en/3.1/topics/http/urls/

[←23]
https://docs.djangoproject.com/en/3.1/ref/urls/#django.urls.include

[←24]
https://docs.djangoproject.com/en/3.1/topics/http/views/#django.http.Http404

[←25]
https://docs.djangoproject.com/en/3.1/ref/models/instances/#django.db.models.Model.get_absolute_url

[←26]
https://docs.djangoproject.com/en/3.1/ref/models/fields/#datefield

[←27]
https://docs.djangoproject.com/en/3.1/ref/csrf/

[←28]
https://docs.djangoproject.com/en/3.1/topics/forms/#working-with-form-templates

[←29]
https://docs.djangoproject.com/en/3.1/topics/http/sessions/

[←30]
https://docs.djangoproject.com/en/3.1/ref/settings/#settings

[←31]
https://docs.djangoproject.com/en/3.1/topics/http/sessions/#when-sessions-are-saved

[←32]
https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#floatformat

[←33]
https://docs.djangoproject.com/en/3.1/ref/templates/builtins/#length

[←34]
https://docs.djangoproject.com/en/3.1/ref/templates/api/#writing-your-own-context-processors

[←35]
https://docs.djangoproject.com/en/3.1/ref/models/fields/#datefield

[←36]
https://stripe.com/docs/payments/payment-intents/migration/charges

[←37]
https://stripe.com/docs/testing#cards

[←38]
https://dashboard.stripe.com/test/payments

[←39]
https://docs.celeryproject.org/en/stable/index.html

[←40]
https://docs.celeryproject.org/en/stable/django/first-steps-with-django.html

[←41]
https://www.rabbitmq.com/

[←42]
https://redis.io/

[←43]
https://docs.celeryproject.org/en/stable/userguide/calling.html

[←44]
https://developers.google.com/gmail/imap/imap-smtp

[←45]
https://support.microsoft.com/en-us/office/pop-imap-and-smtp-settings-for-outlook-com-d088b986-291d-42b8-9564-9c414e2aa040

[←46]
https://www.mailgun.com/

[←47]
https://mailchimp.com/

[←48]
https://weasyprint.readthedocs.io/en/stable/index.html

[←49]
https://docs.djangoproject.com/en/3.1/topics/auth/

[←50]
https://docs.djangoproject.com/en/3.1/ref/models/class/#doesnotexist

[←51]
https://docs.djangoproject.com/en/3.1/topics/auth/default/#authenticating-users

[←52]
https://docs.djangoproject.com/en/3.1/topics/auth/default/#django.contrib.auth.login

[←53]
https://docs.djangoproject.com/en/3.1/ref/contrib/messages/

[←54]
https://docs.djangoproject.com/en/3.1/topics/auth/default/#django.contrib.auth.views.LoginView

[←55]
https://docs.djangoproject.com/en/3.1/ref/contrib/auth/#anony moususer-object

[←56]
https://docs.djangoproject.com/en/3.1/topics/auth/customizing/#using-a-custom-user-model-when-starting-a-project

[←57]
https://docs.djangoproject.com/en/3.1/topics/auth/customizing/#django.contrib.auth.get_user_model

[←58]
https://www.digitalocean.com/

[←59]
https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-20-04

[←60]
https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-20-04

[←61]
https://gunicorn.org/

[←62]
https://www.nginx.com/

[←63]
https://m.do.co/c/36d391016ef7

[←64]
https://www.namecheap.com/

[←65]
https://certbot.eff.org/

[←66]
https://letsencrypt.org/

Table of Contents