**Chapter 04: Bookstore Project**

Don’t follow the book to create V E with python version 3.xx.x because it does not work the same way on VS code. How to create V E with different python version on VS code:

Using VS Code Command Palette:

Open the Command Palette (Ctrl+Shift+P).

Search for and select "Python: Create Environment".

Choose "Venv".

VS Code should present a list of available interpreters, including Python 3.11.8 if it's installed correctly and on your PATH. Select the Python 3.11.8 interpreter from the list.

A notification will indicate the creation progress, and the environment folder will appear in your workspace.

When V E is installed follow this code:

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

Then activate it:

.venv\Scripts\Activate.ps1

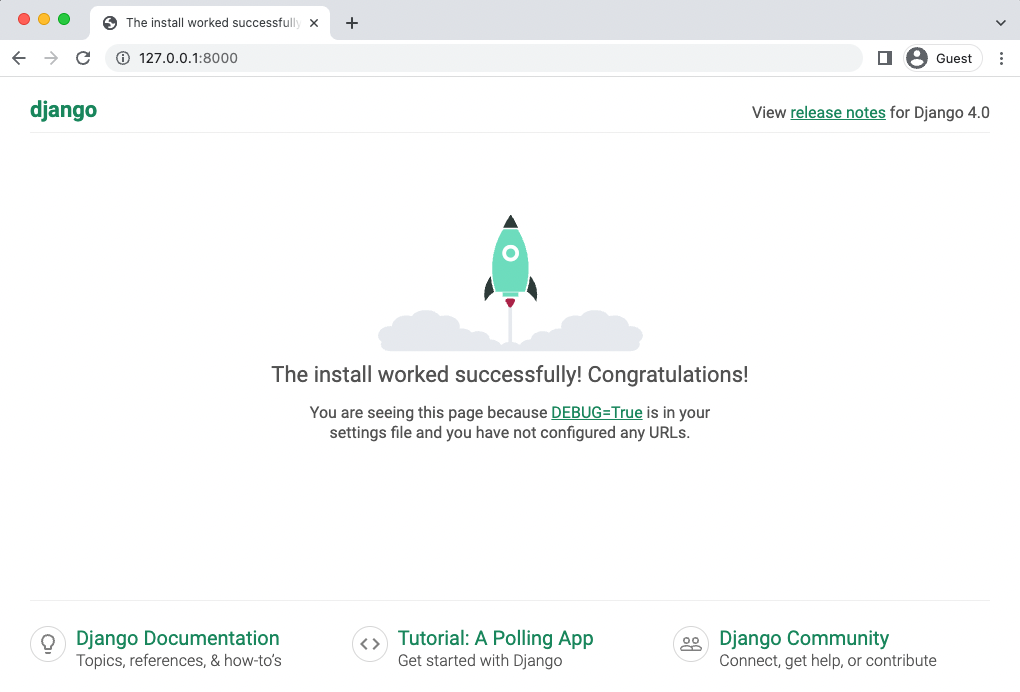
Then install Django, if you need of a specific version of Django use “django~=x.x.x.

We use the startproject command to make our new Django project called django\_project. Make sure you don’t forget that period, ., at the end of the command or else Django will create an extra directory which we don’t need. Then use runserver to start the local Django web server and confirm everything is working correctly.

(.venv) django-admin startproject django\_project .

(.venv) python manage.py runserver

In your web browser go to http://127.0.0.1:8000/ and you should see the friendly Djangowelcome page.



On the command line you will likely see a warning about “18 unapplied migration(s)”. It’s safe to ignore this for now since we’re about to switch over to Docker and PostgreSQL.

Create a requirements.txt with the syntax:

pip freeze > requitements.txt

The new file should appear on vs code explorer and get the following:

asgiref==3.8.1

Django==4.0.10

sqlparse==0.5.3

tzdata==2025.2

psycopg2-binary==2.9.3 (add manually don’t try to install it, it doesn’t work)

**Docker**

We can now switch over to Docker in our project. Go ahead and stop the local server Control+c and also exit the virtual environment shell.

(.venv) deactivate

Next we will need a Dockerfile, docker-compose.yml, .dockerignore and .gitignore, all should be created near to manage.py.

Dockerfile:

# Pull base image

FROM python:3.11.8-slim-bullseye

# Set environment variables

ENV PIP\_DISABLE\_PIP\_VERSION\_CHECK=1

ENV PYTHONDONTWRITEBYTECODE=1

ENV PYTHONUNBUFFERED=1

# Set work directory

WORKDIR /code

# 🔧 Install PostgreSQL dev headers and gcc for psycopg2

RUN apt-get update && apt-get install -y libpq-dev gcc

# Install dependencies

COPY ./requirements.txt .

RUN pip install -r requirements.txt

# Copy project files

COPY . .

Docker-compose.yml:

services:

web:

build: .

command: python /code/manage.py runserver 0.0.0.0:8000

volumes:

- .:/code

ports:

- 8000:8000

depends\_on:

- db

db:

image: postgres:13

volumes:

- postgres\_data:/var/lib/postgresql/data/

environment:

- POSTGRES\_HOST\_AUTH\_METHOD=trust

volumes:

postgres\_data:

Warning : be very careful with the indent

.dockerignore:

.venv

.git

.gitignore

.gitignore:

.venv

\_\_pycache\_\_/

db.sqlite3

.DS\_Store # Mac only

Now we can build our image and run the containers with one command.

docker-compose up -d --build

Go to the web browser now at http://127.0.0.1:8000/ and click refresh. It should be the same friendly Django welcome page albeit now running inside of Docker.

Note: To use terminal again, after docker is done building exit it with control+C and run

docker-compose up -d

Now you can use your terminal and docker is running in background.

**A couple things to check or try next:**

1. Docker-compose version warning:

Your docker-compose.yml has the version: "3.9" line. The warning says this attribute is obsolete, so you can try removing the version: "3.9" line entirely (the newer Docker Compose versions don’t need this anymore).

2. Confirm container ports with Docker CLI:

Run this outside the container:

docker port chapter4bookstoreproject-web-1

It should output something like:

8000/tcp -> 0.0.0.0:8000

This confirms the port mapping is active.

3. Test container network from host:

Try to curl from your host machine (not inside container):

curl http://localhost:8000/

If this works, your Django server is running and accessible.

4. Confirm container is running and logs:

If you want to see logs while detached:

docker-compose logs -f web

This tails the logs and you can Ctrl+C to stop.

Verify firewall or security software blocking connections on port 8000 on your host machine (Windows firewall, antivirus, etc.).

Also: Your Dockerfile installs libpq-dev and gcc which is correct for psycopg2. Make sure your requirements.txt includes psycopg2-binary or psycopg2.

If after those checks it still fails, let me know exactly what you get when you:

run docker port chapter4bookstoreproject-web-1

try curl http://localhost:8000/ from your host terminal

if an error occur do this:

✅ The Fix: Reset Docker’s Database Volume and Start Fresh

This clears any mismatch between code and DB.

⚠️ Warning: This will delete your Postgres DB volume (dev-only — safe if you're following a tutorial).

🧹 1. Stop and Remove All Containers and Volumes

docker-compose down -v

This stops everything and removes all volumes, including the PostgreSQL database.

🔁 2. Rebuild and Restart Everything

bash

docker-compose up -d --build

🛠️ 3. Reapply Migrations

docker-compose exec web python manage.py makemigrations

docker-compose exec web python manage.py migrate

At this point you should not see:

“Dependency on app with no migrations”

“No changes detected” incorrectly

or a broken server loop

**Custom User Model**

Now we come to new material! We begin by implementing a custom user model which the official Django documentation “highly recommends.” Why? Because you will need to make changes to the built-in User model at some point in your project’s life and if you have not started with a custom user model from the very first migrate command you run, then you’re in for a world of hurt because User is tightly interwoven with the rest of Django internally. It is challenging to switch over to a custom user model mid-project.

A point of confusion for many people is that custom user models were only added in Django 1.5. Up until that point the recommended approach was to add a OneToOneField, often called a Profile model, to User. You’ll often see this set up in older projects.

But these days using a custom user model is the more common approach. However as with many things Django-related, there are implementation choices: either extend AbstractUser which keeps the default User fields and permissions or extend AbstractBaseUser which is even more granular, and flexible, but requires more work.

We’ll stick with the simpler AbstractUser in this book as AbstractBaseUser can be added later if needed.

There are four steps for adding a custom user model to our project:

1. Create a CustomUser model

2. Update django\_project/settings.py

3. Customize UserCreationForm and UserChangeForm

4. Add the custom user model to admin.py

The first step is to create a CustomUser model which will live within its own app. I like to name this app accounts. Going forward we’ll run the majority of our commands within Docker itself. Because we configured a volumes within the web service of docker-compose.yml any file system changes within Docker will be reflected in the local file system.

docker-compose exec web python manage.py startapp accounts

Create a new CustomUser model which extends AbstractUser. That means we’re essentially making a copy where CustomUser now has inherited all the functionality of AbstractUser, but wecan override or add new functionality as needed. We’re not making any changes yet so include the Python pass statement which acts as a placeholder for our future code.

# accounts/models.py

from django.contrib.auth.models import AbstractUser

from django.db import models

class CustomUser(AbstractUser):

pass

Now go in and update our settings.py

# django\_project/settings.py

INSTALLED\_APPS = [

"django.contrib.admin",

"django.contrib.auth",

"django.contrib.contenttypes",

"django.contrib.sessions",

"django.contrib.messages",

"django.contrib.staticfiles",

# Local

"accounts.apps.AccountsConfig", # new

]

...

AUTH\_USER\_MODEL = "accounts.CustomUser" # new (bottom of setting)

Time to create a migrations file for the changes. We’ll add the optional app name accounts to the command so that only changes to that app are included.

docker-compose exec web python manage.py makemigrations accounts

**Custom User Forms**

A user model can be both created and edited within the Django admin. So we’ll need to update the built-in forms too to point to CustomUser instead of User. In your text editor create a new file called accounts/forms.py and type in the following code to switch over to CustomUser

# accounts/forms.py

from django.contrib.auth import get\_user\_model

from django.contrib.auth.forms import UserCreationForm, UserChangeForm

class CustomUserCreationForm(UserCreationForm):

class Meta:

model = get\_user\_model()

fields = (

"email",

"username",

)

class CustomUserChangeForm(UserChangeForm):

class Meta:

model = get\_user\_model()

fields = (

"email",

"username",

)

At the very top we’ve imported CustomUser model via get\_user\_model which looks to our AUTH\_USER\_MODEL config in settings.py. This might feel a bit more circular than directly importing CustomUser here, but it enforces the idea of making one single reference to the custom user model rather than directly referring to it all over our project.

Next we import UserCreationForm and UserChangeForm which will both be extended.

Then create two new forms–CustomUserCreationForm and CustomUserChangeForm–that extend the base user forms imported above and specify swapping in our CustomUser model and displaying the fields email and username. The password field is implicitly included by default and so does not need to be explicitly named here as well.

**Custom User Admin**

Finally we have to update our accounts/admin.py file. The admin is a common place to

manipulate user data and there is tight coupling between the built-in User and the admin.

We’ll extend the existing UserAdmin into CustomUserAdmin and tell Django to use our new forms and custom user model. We can also list any user attributes we want but for now will just focus on three: email, username, and superuser status.

from django.contrib import admin

from django.contrib.auth import get\_user\_model

from django.contrib.auth.admin import UserAdmin

from .forms import CustomUserCreationForm, CustomUserChangeForm

CustomUser = get\_user\_model()

# Register your models here.

class CustomUserAdmin(UserAdmin):

add\_form = CustomUserCreationForm

form = CustomUserChangeForm

model = CustomUser

list\_display = [

"email",

"username"

"is\_superuser",

]

admin.site.register(CustomUser, CustomUserAdmin)

Phew. A bit of code upfront but this saves a ton of heartache later on.

**Superuser**

A good way to confirm our custom user model is up and running properly is to create a superuser account so we can log into the admin. This command will access CustomUserCreationForm under the hood.

docker-compose exec web python manage.py createsuperuser

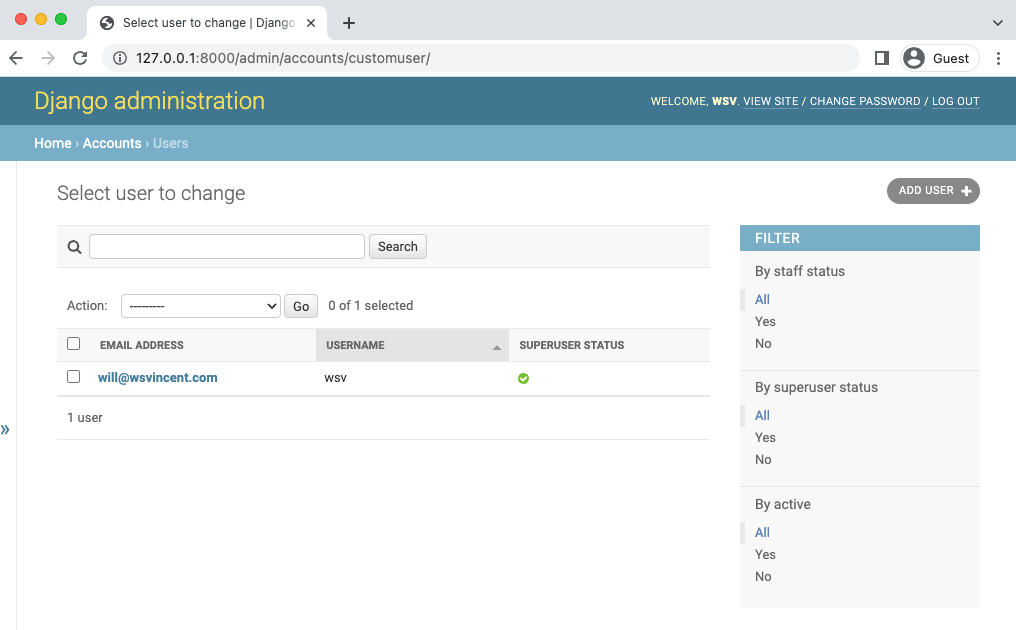
I’ve used the username wsv, email of will@wsvincent.com, and password testpass123. You can use your own preferred variations here.

your web browser open the page http://127.0.0.1:8000/admin and log in. You should see your superuser name in the upper right corner on the post-log in page.



Django admin homepage

You can also click on the Users section to see the email and username of your superuser account.



Django admin users page

**Tests**

Since we’ve added new functionality to our project we should test it. Whether you are a solo developer or working on a team, tests are important. In the words of Django co-founder Jacob Kaplan-Moss, “Code without tests is broken as designed.”

There are two main types of tests:

• Unit tests are small, fast, and isolated to a specific piece of functionality

• Integration tests are large, slow, and used for testing an entire application or a user flow like payment that covers multiple screens

You should write many unit tests and a small number of integration tests.

The Python programming language contains its own unit testing framework and Django’s automated testing framework extends this with multiple additions into a web context. There is no excuse for not writing a lot of tests; they will save you time.

It’s important to note that not everything needs to be tested. For example, any built-in Django features already contain tests in the source code. If we were using the default User model in our project we would not need to test it. But since we’ve created a CustomUser model we should.

**Unit Tests**

To write unit tests in Django we use TestCase which is, itself, an extension of Python’s TestCase. Our accounts app already contains a tests.py file which is automatically added when the startapp command is used. Currently it is blank. Let’s fix that!

Each method must be prefaced with test in order to be run by the Django test suite. It is also a good idea to be overly descriptive with your unit test names since mature projects have hundreds if not thousands of tests!

# accounts/tests.py

from django.contrib.auth import get\_user\_model

from django.test import TestCase

# Create your tests here.

class CustomUserTests(TestCase):

def test\_create\_user(self):

User = get\_user\_model()

user = User.objects.create\_user(

username="jim57",

email="jim57@example.com",

password="testpass123"

)

self.assertEqual(user.username, "jim57")

self.assertEqual(user.email, "jim57@example.com")

self.assertTrue(user.is\_active)

self.assertFalse(user.is\_staff)

self.assertFalse(user.is\_superuser)

def test\_create\_superuser(self):

User = get\_user\_model()

admin\_user = User.objects.create\_superuser(

username="superadmin",

email="superadmin@email.com",

password="testpass123"

)

self.assertEqual(admin\_user.username, "superadmin")

self.assertEqual(admin\_user.email, "superadmin@email.com")

self.assertTrue(admin\_user.is\_active)

self.assertTrue(admin\_user.is\_staff)

self.assertTrue(admin\_user.is\_superuser)

We have imported both get\_user\_model and TestCase before creating a CustomUserTests class. Within it are two separate tests. test\_create\_user confirms that a new user can be created. First we set our user model to the variable User and then create one via the manager method create\_user which does the actual work of creating a new user with the proper permissions.

For test\_create\_superuser we follow a similar pattern but reference create\_superuser instead of create\_user. The difference between the two users is that a superuser should have both is\_staff and is\_superuser set to True.

To run our tests within Docker we’ll prefix docker-compose exec web to the traditional command python manage.py test.

docker-compose exec web python manage.py test

Found 2 test(s).

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

..

---------------------------------------------------------------------

Ran 2 tests in 0.115s

OK

Destroying test database for alias 'default'...

All the tests pass so we can proceed.

**Git**

We’ve accomplished quite a lot in this chapter so it is a good point to pause and commit our work

by initializing a new Git repository, adding changes, and including a commit message.

git init

git status

git add .

git commit -m “Chapter 04. Bookstore Project”

**To push to GitHub if you have an account:**

1. run: cd ..

Your path should be where you root depository is(in my case Bookstore-by-Chapter):

PS C:\Users\*computer’s username*\Documents\*your main folder*\Bookstore-by-Chapter>

Yours would be different like:

PS C:\Users\*computer’s username*\Documents\*your main folder*\title of your subfolder>

git add " Chapter 04. Bookstore Project"

git commit -m " Chapter 04. Bookstore Project"

git push

You can compare with the official source code for this chapter on Github.

**Conclusion**

Our Bookstore project is now running with Docker and PostgreSQL and we’ve configured a custom user model. Next up will be a pages app for our static pages.

The end