**Chapter 07: Static Assets**

Static assets like CSS, JavaScript, and images are a core component of any modern web application. Django provides us with a large degree of flexibility around their configuration and storage which can be confusing to newcomers but in practice the pattern is almost identical for all projects. In this chapter we’ll configure our initial static assets and add Bootstrap for improved styling.

**Local Development**

For local development the Django web server automatically serves static files and minimal configuration is required. Static files can be placed in an app-level directory called static so we could, for example, create a new directory within the pages app called pages/static/.

However most projects reuse static assets across multiple apps. Our bookstore app certainly will! So a more common approach is to create a base-level directory static folder and place all static files within there. This is easier to reason about as a developer.

From the command line create a new static folder along with new subdirectories for CSS, JavaScript, and images.

mkdir static

mkdir static/css

mkdir static/js

mkdir static/images

Your Django project should have the following structure at this point:

**Project Structure**

├── Dockerfile

├── accounts

│ ...

├── db.sqlite3

├── django\_project

│ ...

├── docker-compose.yml

├── manage.py

├── pages

│ ...

├── requirements.txt

├── static

│ ├── css

│ ├── images

│ └── js

└── templates

...

It’s important to be aware that Git, by default, will not track directories that are empty. Thus when we eventually push our code to both GitHub and Heroku these empty directories will not appear which can cause problems in deployment when collectstatic is run. To avoid this we will add an empty file to each directory now.

In your text editor create the following three files:

• static/css/base.css

• static/js/base.js

• static/images/.keep

**STATICFILES\_DIRS**

At the bottom of the django\_project/settings.py file is a section on “Static files” which has already set STATIC\_URL to "static/". This is the URL used when referring to static files. It means that for local usage all static files are located at <http://127.0.0.1:8000/static/>.

The built-in staticfiles app that ships with Django–you can see it in the INSTALLED\_APPS section of django\_project/settings.py–ships with a quick and dirty helper view that will serve files locally for development. It will automatically look for a static directory within each application.

Many static files are not app-specific though and are intended to be used project-wide. That’s why creating a base directory static folder, as we did above, is a common practice. We just need to provide a list of additional directories for Django to look in, which is the job of STATICFILES\_DIRS. Since this is a list we must place brackets, [ ], around it in standard Python syntax.

Here is what the updated django\_project/settings.py file should look like.

# django\_project/settings.py

STATIC\_URL = "/static/"

STATICFILES\_DIRS = [BASE\_DIR / "static"] # new

**CSS**

We’ll update base.css now and keep things simple by having our h1 headline be red. The pointis to show how CSS can be added to our project, not to delve too deeply into CSS itself.

/\* static/css/base.css \*/

h1 {

color: red;

}

If you refresh the homepage now you’ll see that nothing has changed. That’s because static assets must be explicitly loaded into templates! To do this we must add {% load static %} at the top of \_base.html and then use the static template tag to reference our base.css file. Even though this file is located at static/css/base.css we can refer to it as css/base.css because the static tag automatically looks within the /static/ directory specified in STATIC\_URL.

Here is what the updated templates/\_base.html file should look like.

<!-- templates/\_base.html -->

{% load static %}

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>{% block title %}Bookstore{% endblock %}</title>

<!-- CSS -->

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

</head>

...

Refresh the homepage to see our work.



Homepage with red text

There’s our CSS in action! If instead you see an error screen saying Invalid block tag on line 7: 'static'. Did you forget to register or load this tag? then you forgot to include the line {% load static %} at the top of the file. I do this all the time myself.

**Images**

How about an image? You can download the book cover for Django for Professionals at this link. Save it into the directory books/static/images as dfp.png.

To display it on the homepage, update templates/home.html by adding the {% load static %} tag at the top. Then under the <h1> tags add an <img> class that also uses static to display the new cover.

<!-- templates/home.html -->

{% extends "\_base.html" %}

{% load static %} # new

{% block title %}Home{% endblock title %}

{% block content %}

<h1>This is our home page.</h1>

<img class="bookcover" src="{% static 'images/dfp.png' %}"> #new

{% if user.is\_authenticated %}

<p>Hi {{ user.email }}!</p>

<p><a href="{% url 'logout' %}">Log Out</a></p>

{% else %}

<p>You are not logged in</p>

<p><a href="{% url 'login'%}">Log In</a> |

<a href="{% url 'signup' %}">Sign Up</a></p>

{% endif %}

{% endblock content %}

Refreshing the homepage you’ll see the raw file is quite large! Let’s control that with some additional CSS.

/\* static/css/base.css \*/

h1 {

color: red;

}

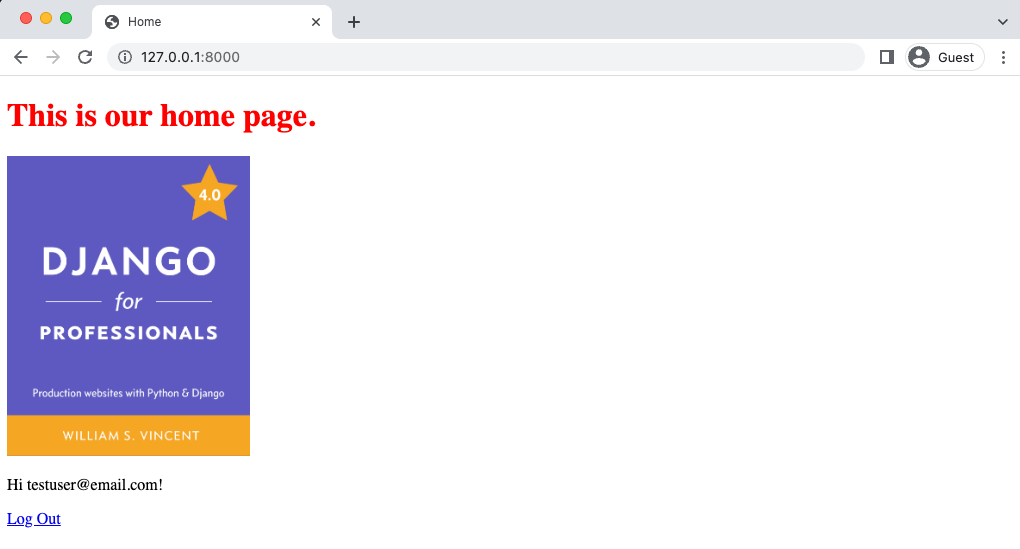
.bookcover {

height: 300px;

width: auto;

}

Now refresh the homepage and the book cover image fits nicely.



Homepage with Book Cover

If you don’t see the change you might need to perform a “hard refresh” to bypass your web browser’s local cache. Modern web browsers automatically keep local copies of pages, images, and other content already visited to speed up performance. On occasion this automatic caching results in a browser showing out-of-date content. The solution is force a re-download of a web page’s content, also known as a “hard refresh” or “cache refresh”. On Windows this can be down by holding Ctrl+F5; on macOS hold Cmd+Shift+R.

**JavaScript**

We already have a base.js file we can use to add JavaScript to our project. Often I put a tracking code of some kind here, such as for Google Analytics, but for demonstration purposes we’ll add a console.log statement so we can confirm the JavaScript loaded correctly.

// static/js/base.js

console.log("JavaScript here!")

Now add it to our \_base.html template. JavaScript should be added at the bottom of the file so it is loaded last, after the HTML, CSS, and other assets that appear first on the screen when rendered in the web browser. This gives the appearance of the complete webpage loading faster.

<!-- templates/\_base.html -->

{% load static %}

<!DOCTYPE html>

<html>

<head>

<meta cahrset="utf-8">

<title>{% block title %}Bookstore{% endblock title %}</title>

<!-- CSS -->

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

</head>

<body>

<div class="container">

{% block content %}

{% endblock content %}

</div>

<!-- JavaScript -->

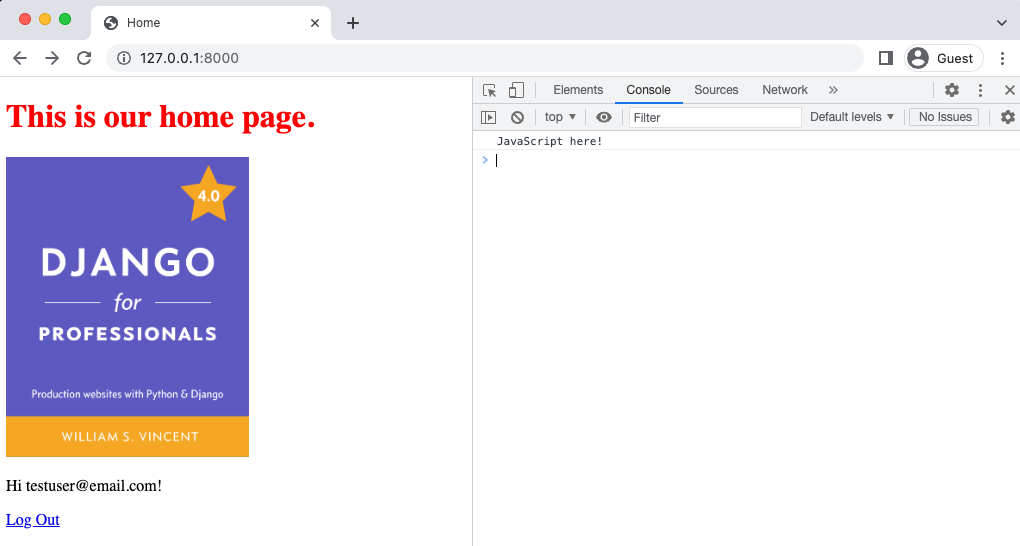
<script src="{% static 'js/base.js' %}"></script>

</body>

</html>

In your web browser, make the JavaScript console available. This involves opening up Developer Tools and making sure you’re on the “Console” section. On Chrome which is being used for the images in this book, go to View in the top menu, then Developer -> Developer Tools which willopen a sidebar. Make sure Console is selected from the options.

If you refresh the page, you should see the following:



Homepage JavaScript console view

**Production**

Local development is designed to keep things nice and easy for static files. Whether they are located in an individual app or in a base-level directory, the Django server will display everything for us. But this approach is not very efficient since each resource must be individually requested and served from each location. In a production environment it is far more efficient to combine all static files into one location and serve that in a single, larger HTTP request.

Django comes with a built-in management command, collectstatic, that does this for us. But we need to configure two more settings in our django\_project/settings.py file before collectstatic will work properly.

The first setting is STATIC\_ROOT which sets the absolute location of these collected files. This can be named anything you like but is typically called “staticfiles”. When collecstatic is run locally it will combine all available static files as defined by STATICFILES\_DIRS and place them within a directory as defined by STATIC\_ROOT. We are setting STATIC\_ROOT to be in the base directory with the name staticfiles. Here is what the code looks like:

# django\_project/settings.py

STATIC\_URL = "/static/"

STATICFILES\_DIRS = [BASE\_DIR / "static"]

STATIC\_ROOT = BASE\_DIR / "staticfiles" # new

The second setting is STATICFILES\_STORAGE, which is the file storage engine used when collecting static files with the collectstatic command. By default, it is implicitly set to django.contrib.staticfiles.storage.StaticFilesStorage. Let’s make that explicit for now in our django\_project/settings.py file.

# django\_project/settings.py

STATIC\_URL = "/static/"

STATICFILES\_DIRS = [BASE\_DIR / "static"]

STATIC\_ROOT = BASE\_DIR / "staticfiles"

STATICFILES\_STORAGE = "django.contrib.staticfiles.storage.StaticFilesStorage"#new

There is actually one more default static files setting available, STATICFILES\_FINDERS, but it is typically not customized so we will leave it out of the django\_project/settings.py file at this time.

Now we can run the command python manage.py collectstatic which will combine all static files into a new staticfiles directory.

docker-compose exec web python manage.py collectstatic

131 static files copied to '/code/staticfiles'.

If you look within your text editor, there is now a staticfiles directory with four subdirectories: admin, css, images, and js. The first one is the static assets of the Django admin app and the other three we specified. That’s why there are 122 files copied over.

**Bootstrap**

Writing custom CSS for your website is a worthy goal and something I advise all software developers, even back-end ones, to try at some point. But practically speaking there is a reason front-end frameworks like Bootstrap exist: they save you a ton of time when starting a new project. Unless you have a dedicated designer to collaborate with, stick with a framework for the early iterations of your website.

Bootstrap can be installed locally or used via a CDN. The latter is a far simpler approach so we will adopt it in this book. Instructions can be found here to deliver a cached version of Bootstrap’s compiled CSS and JS to our project.

that the full links for Bootstrap are not included. You should copy and paste them in from the Bootstrap website to be accurate. They are referenced here as Bootstrap CSS and Bootstrap JavaScript.

{% load static %}

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>{% block title %}Bookstore{% endblock title %}</title>

<!-- Bootstrap CSS -->

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">

<!-- CSS -->

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

</head>

<body>

<div class="container">

{% block content %}

{% endblock content %}

</div>

<!-- Bootstrap Javascript -->

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">

<!-- JavaScript -->

<script src="{% static 'js/base.js' %}"></script>

</body>

</html>

Note that order matters here for both the CSS and JavaScript. The file will be loaded top-tobottom so our base.css file comes after the Bootstrap CSS. This means our custom h1 style will override the Bootstrap default. At the bottom of the file we also place our base.js file after the Bootstrap JavaScript file.

Now it’s time to actually use Bootstrap. We’ll start by adding a fixed navbar to the \_base.html file at the top of the <body> section.

<!-- templates/\_base.html -->

…

<body>

<nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">

<div class="container-fluid">

<a class="navbar-brand" href="#">Bookstore</a>

<button class="navbar-toggler" type="button"

data-bs-toggle="collapse"

data-bs-target="#navbarCollapse"

aria-controls="navbarCollapse"

aria-expanded="false"

aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarCollapse">

<ul class="navbar-nav me-auto mb-2 mb-md-0">

{% if user.is\_authenticated %}

<li class="nav-item">

<a class="nav-link" href="{% url 'logout' %}">Log Out</a>

</li>

{% else %}

<li class="nav-item">

<a class="nav-link" href="{% url 'login' %}">Log In</a>

</li>

<li class="nav-item">

<a class="nav-link" href="{% url 'signup' %}">Sign Up</a>

</li>

{% endif %}

</ul>

</div>

</div>

</nav>

<div class="container">

{% block content %}

…

The other step is to update base.css so that our body section fits the new navbar.

/\* static/css/base.css \*/

body {

min-height: 75rem;

padding-top: 4.5rem;

}

h1 {

color: red;

}

img.bookcover {

height: 300px;

width: auto;

}

The code in our home.html template can be cleaned up now too. We don’t need to display Log In or Sign Up links. Nor do we need to display a message about the user not being logged in. The reduced home.html template should look as follows:

<!-- templates/home.html -->

{% extends "\_base.html" %}

{% load static %}

{% block title %}Home{% endblock title %}

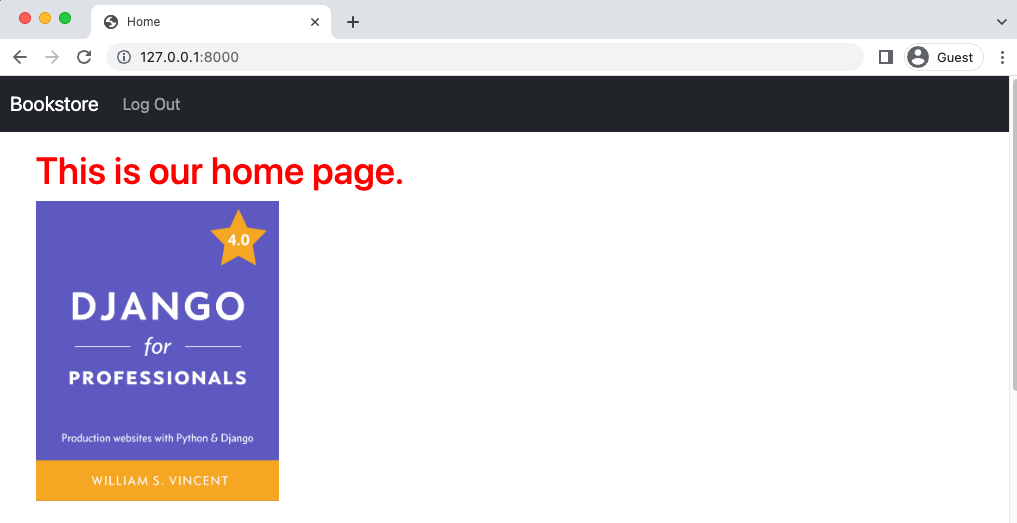
{% block content %}

<h1>This is our home page.</h1>

<img class="bookcover" src="{% static 'images/dfp.png' %}">

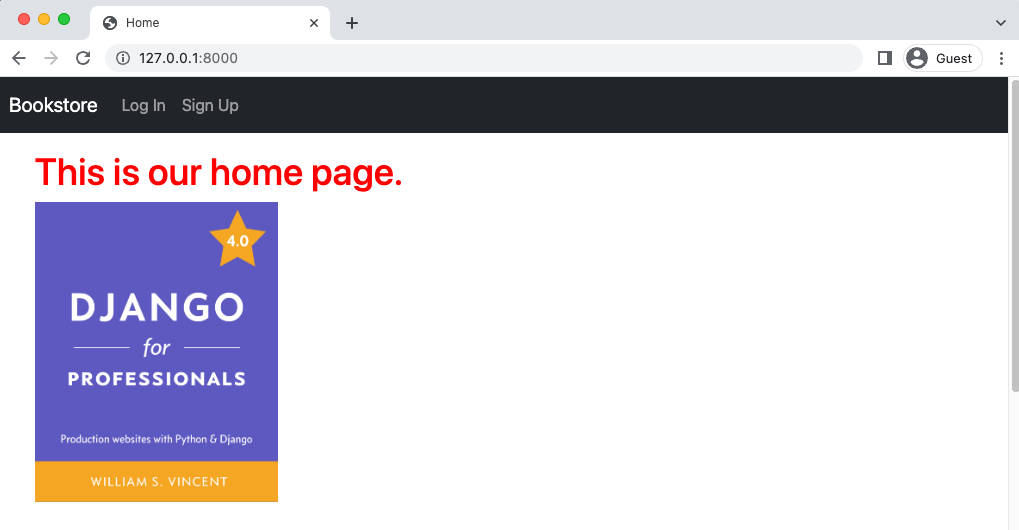
{% endblock content %}

Refresh the homepage now to see our new navbar and updated homepage. The web page body just displays the title and book cover.



Homepage with Bootstrap

If you click the “Log Out” link in the navbar it automatically updates it to instead include “Log In” and “Sign Up”.



Homepage with Bootstrap Logged Out

**About Page**

Let’s add an About page to our project now and to the navbar. It is quite straightforward to do now that we have our site’s scaffolding configured. Since this will be a static page we don’t need a database model involved. However we will need a template, view, and url.

Let’s start by creating a new template file called templates/about.html. The page will literally just say “About Page” for now while inheriting from \_base.html.

<!-- templates/about.html -->

{% extends "\_base.html" %}

{% block title %}About{% endblock title %}

{% block content %}

<h1>About Page</h1>

{% endblock content %}

The view can rely on Django’s built-in TemplateView just like our homepage.

# pages/views.py

from django.views.generic import TemplateView

# Create your views here.

class HomePageView(TemplateView):

template\_name = "home.html"

class AboutPageView(TemplateView): # new

template\_name = "about.html"

And the URL path will be familiar as well. In the pages/urls.py file we will first import the

necessary view, AboutPageView, create a new URL path to about/, and provide a URL name of about.

# pages/urls.py

from django.urls import path

from .views import HomePageView, AboutPageView # new

urlpatterns = [

path("about/", AboutPageView.as\_view(), name="about"), # new

path("", HomePageView.as\_view(), name="home"),

]



About Page

As a final step, update the link in the navbar to the page. Because we provided a name in the URL path of about that’s what we’ll use alongside the url template tag. On line 18 of \_base.html change the line with the About page link to the following:

<!-- templates/\_base.html -->

…

<div class="collapse navbar-collapse" id="navbarCollapse">

<ul class="navbar-nav me-auto mb-2 mb-md-0">

<li class="nav-item">

<a class="nav-link" href="{% url 'about'}">About</a>

</li>

…

If you refresh the about page now it will display the “About” link in the navbar.



About Navbar Link

There has been a lot of code so far in this chapter especially in the \_base.html file. If you are concerned about a small typo it is more than fine to look at the official source code here and just copy and paste the templates/\_base.html file if needed.

**About Page Tests**

Time for tests. We’ve added a new About page so we should add basic tests for it in pages/tests.py. These tests will be very similar to those we added previously for our homepage.

# pages/tests.py

from django.test import SimpleTestCase

from django.urls import reverse, resolve # new

from .views import HomePageView, AboutPageView # new

# Create your tests here.

class HomepageTests(SimpleTestCase):

…

class AboutPageTest(SimpleTestCase): # new

def setUp(self):

url = reverse("about")

self.response = self.client.get(url)

def test\_aboutpage\_status\_code(self):

self.assertEqual(self.response.status\_code, 200)

def test\_aboutpage\_template(self):

self.assertTemplateUsed(self.response, "about.html")

def test\_aboutpage\_contains\_correct\_html(self):

self.assertContains(self.response, "About Page")

def test\_aboutpage\_does\_not\_contain\_incorrect\_html(self):

self.assertNotContains(

self.response, "Hi there! I should not be on the page.")

def test\_aboutpage\_url\_resolves\_aboutpageview(self):

view = resolve("/about/")

self.assertEqual(view.func.\_\_name\_\_, AboutPageView.as\_view().\_\_name\_\_)

Run the tests.

docker-compose exec web python manage.py test

Found 15 test(s).

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

...............

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

Ran 15 tests in 0.161s

OK

Destroying test database for alias 'default'...

They all passed! Good. Now we can feel confident that any future changes we make to our project won’t mess up the About page.

**Django Crispy Forms**

One last update with Bootstrap is to our forms. The popular 3rd party package django-crispyforms provides a host of welcome upgrades and has a Bootstrap5 dedicated template pack crispybootstrap5 that we can use too. Add both to our requirements.txt file as follows:

requirements.txt

asgiref==3.5.2

Django==4.0.4

psycopg2-binary==2.9.3

sqlparse==0.4.2

django-crispy-forms==1.14.0

crispy-bootstrap5==0.6

Then stop our Docker container and rebuild it so that django-crispy-forms and crispy-bootstrap5

are available.

docker-compose down

docker-compose up –build -d

Our Dockerfile has a command to check requirements.txt to see if anything has changed and, if so, to install it in our Docker image which is then run by our container as spelled out in docker-compose.yml. This is a safe way to install new packages when working with Docker.

Once installed we must add django-crispy-forms and crispy\_bootstrap5 to the INSTALLED\_APPS setting. Note that the app name needs to be crispy\_forms here! A nice additional feature is to specify bootstrap4 under CRISPY\_TEMPLATE\_PACK which will provide pre-styled forms for us.

# django\_project/settings.py

INSTALLED\_APPS = [

"django.contrib.admin",

"django.contrib.auth",

"django.contrib.contenttypes",

"django.contrib.sessions",

"django.contrib.messages",

"django.contrib.staticfiles",

# Third-party

"crispy\_forms", # new

"crispy\_bootstrap5", # new

# Local

"accounts.apps.AccountsConfig",

"pages.apps.PagesConfig",

]

# django-crispy-forms

CRISPY\_ALLOWED\_TEMPLATE\_PACKS = "bootstrap5" # new

CRISPY\_TEMPLATE\_PACK = "bootstrap5" # new

To use Crispy Forms we load crispy\_forms\_tags at the top of a template and add {{ form|crispy }} to replace {{ form.as\_p}} for displaying form fields. We will take this time to also add Bootstrap styling to the Submit button.

Start with the templates/registration/signup.html file and make the updates below.

<!-- templates/registration/signup.html -->

{% extends "\_base.html" %}

{% load crispy\_forms\_tags %} # new

{% block title %}Sign Up{% endblock title %}

{% block content %}

<h2>Sign Up</h2>

<form method="post">

{% csrf\_token %}

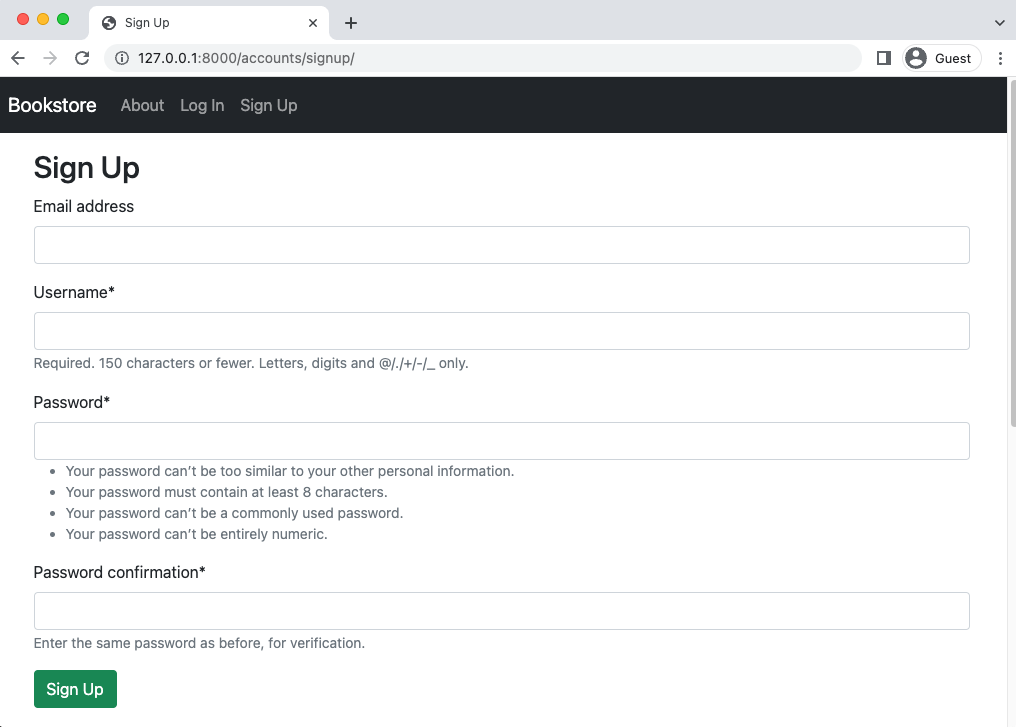
{{ form|crispy }}

<button class="btn btn-success" type="submit">Sign Up</button> # new

</form>

{% endblock content %}

Then navigate to the sign up page at <http://127.0.0.1:8000/accounts/signup/>.



Sign Up Page with Crispy Forms

Next update templates/registration/login.html as well with crispy\_forms\_tags at the top

and {{ form|crispy }} in the form.

<!-- templates/registration/login.html -->

{% extends "\_base.html" %}

{% load crispy\_forms\_tags %} # new

{% block title %}Log{% endblock title %}

{% block content %}

<h2>Log In</h2>

<form method="post">

{% csrf\_token %}

{{ form|crispy }}

<button class="btn btn-success" type="submit">Log In</button> # new

</form>

{% endblock content %}

And navigate to the log in page at <http://127.0.0.1:8000/accounts/login/>.



Log In Page with Crispy Forms

**Git**

Wrapping up we should always do a Git commit for our work. Check the status of changes in

this chapter, add them all, and then provide a commit message.

git init

git status

git add .

git commit -m "Chapter 07. Static Assets"

**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 07. Static Assets"

git commit -m "Chapter 07. Static Assets"

git push

As alway you can compare your code with the official code on Github if there are any issues.

**Create a back up:**

Copy-Item -Recurse -Path "C:\Users\Jean-Marc H\Documents\Django for professionals\Bookstore-by-Chapter\Chapter x. chapter’s title" -Destination "C:\Users\Jean-Marc H\Documents\Django for professionals\Bookstore-by-Chapter\ Chapter x. chapter’s title - Backup"

**Conclusion**

Static assets are a core part of every website and in Django we have to take a number of additional steps so they are compiled and hosted efficiently in production. We also saw how easy it is to add additional static pages, such as our About page, to our existing site. And we added Bootstrap and django-crispy-forms to improve the styling of our site and our forms. Later on in the book we’ll learn how to use a dedicated content delivery network (CDN) for hosting and displaying our project’s static files.

The end.