**Chapter 18: Deployment**

So far we have been working entirely in a local development environment on our computer. But now it is time to deploy our project so that it is accessible to the public. In truth, the topic of deployment is worth an entire book on its own. Compared to other web frameworks Django is very hands-off and agnostic on the topic. There are no one-click deploys for most hosting platforms and while this requires more developer work it also allows, in typical Django fashion, for a high degree of customization.

In the previous chapter we configured a completely separate docker-compose-prod.yml file and updated django\_project/settings.py to be production-ready. In this chapter we’ll review how to choose a hosting provider, add a production-ready web server, and properly configure static/media files before deploying our Bookstore site!

**PaaS vs IaaS**

The first question is whether to use a Platform-as-a-Service (PaaS) or Infrastructure-as-a- Service (IaaS). A PaaS is an opinionated hosting option that handles much of the initial configuration and scaling needed for a website. Popular examples include Heroku, PythonAnywhere, and Dokku among many others. While a PaaS costs more money upfront than an IaaS it saves an incredible amount of developer time, handles security updates automatically, and can be quickly scaled.

An IaaS by contrast provides total flexibility and is typically cheaper, but it requires a high degree of knowledge and effort to properly set up. Prominent IaaS options include DigitalOcean, Linode, Amazon EC2, and Google Compute Engine among many others.

So which one to use? Django developers tend to fall in one of two camps: either they already have a deployment pipeline configured with their IaaS of choice or they use a PaaS. Since the former is far more complex and varies widely in its configuration, we will use a PaaS in this book, specifically Heroku/Render.

The choice of Heroku is somewhat arbitrary, but it is a mature technology that comes with a truly free tier sufficient for deploying our Bookstore project.

**WhiteNoise**

We have already seen that for local development Django relies on the staticfiles app to serve static files from across an entire project. This is convenient, but quite inefficient and likely insecure, too.

For production, the collectstatic command must be run to compile all static files into a single directory specified by STATIC\_ROOT. The consolidated files can then be served either on the same server, a separate server, or a dedicated cloud service/CDN by updating STATICFILES\_STORAGE.

In our project, we will rely on serving files from our server with the aid of the WhiteNoise project which works extremely well on Heroku and is both faster and more configurable than Django defaults. To install whitenoise we will add it to our requirements.txt file, stop the container, rebuild the image, and start the server again.

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

django-allauth==0.50.0

environs[django]==9.5.0

pillow==9.0.1

django-debug-toolbar==3.4.0

whitenoise==6.1.0

Then stop our Docker container.

docker-compose down

We won’t rebuild the image and restart the container just yet because we also have to make changes to our django\_project/settings.py file first. Since we’re using Docker it’s possible to switch to WhiteNoise locally as well as in production. While it’s possible to do this by passing in a --nostatic flag to the runserver command, this becomes tiring in practice. A better approach is to add whitenoise.runserver\_nostatic before django.contrib.staticfiles in the INSTALLED\_APPS config which will do the same thing. We’ll also add it to our MIDDLEWARE right below SecurityMiddleware and update STATICFILES\_STORAGE to use WhiteNoise now.

# django\_project/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'whitenoise.runserver\_nostatic', # new

'django.contrib.staticfiles',

'django.contrib.sites',

…

]

MIDDLEWARE = [

'django.middleware.cache.UpdateCacheMiddleware',

'django.middleware.security.SecurityMiddleware',

'whitenoise.middleware.WhiteNoiseMiddleware' # new

…

]

STATICFILES\_STORAGE =

"whitenoise.storage.CompressedManifestStaticFilesStorage” # new

Note that STATICFILES\_STORAGE should be one line. It is formatted on two lines here to fit the book format.

With all our changes made we can now start up our project again in local development mode.

docker-compose up -d --build

WhiteNoise has additional options to serve compressed content and far-future cache headers on content that won’t change. But for now, go ahead and run the collectstatic command. There will be a warning about overwriting existing files. That’s fine. Type “yes” and then hit the “Return” key to continue.

docker-compose exec web python manage.py collectstatic

...

140 static files copied to '/code/staticfiles', 2 unmodified, 404 post-processed.

**Media Files**

Media files such as the book covers in our site are unfortunately a little more complicated than static files for reasons covered in Chapter 13. As a result, while they will appear as desired in local development, they will not remain long in a production setting. The recommended approach is to use the very popular django-storages package alongside a dedicated CDN like S3. However this requires additional configuration that is beyond the scope of this book.

**Gunicorn**

When we ran the startproject command way back in Chapter 3, a django\_project/wsgi.py

file was created with a default WSGI (Web Server Gateway Interface) configuration. This is a specification for how a web app (like our Bookstore project) communicates with a web server.

For production it is common to swap this out for either Gunicorn or uWSGI. Both offer a

performance boost, but Gunicorn is more focused and simpler to implement so it will be our choice.

First add it to the bottom of the requirements.txt file.

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

django-allauth==0.50.0

environs[django]==9.5.0

pillow==9.0.1

django-debug-toolbar==3.4.0

whitenoise==6.1.0

gunicorn==20.1.0

Because we are using Docker our local environment can mimic production quite easily so we’ll update both docker-compose.yml and docker-compose-prod.yml to use Gunicorn instead of the local server.

docker-compose.yml

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

command: gunicorn django\_project.wsgi -b 0.0.0.0:8000 # new

docker-compose-prod.yml

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

command: gunicorn django\_project.wsgi -b 0.0.0.0:8000 # new

Then spin down our container, rebuild a new image, and restart the container.

docker-compose down

docker-compose up -d –build

And after run docker-compose logs web --tail=50 to make sure the module third-party is there.

ALLOWED\_HOSTS

Now that we know the dedicated URL of our production website we can add it to our django\_project/settings.py file. Update the existing entry for ALLOWED\_HOSTS as follows:

# django\_project/settings.py

ALLOWED\_HOSTS = ["\*"] # new

**Note:** This is temporary and should be replaced with your actual Render URL (e.g., "your-app-name.onrender.com") after deployment for better security.

**SECURE\_PROXY\_SSL\_HEADER**

Render uses proxies, so you must configure Django to trust the X-Forwarded-Proto header that Render sets when forwarding HTTPS requests. By default, SECURE\_PROXY\_SSL\_HEADER is None. Since we trust Render’s proxy, update this setting in your django\_project/settings.py file as follows:

# django\_project/settings.py

SECURE\_PROXY\_SSL\_HEADER = ("HTTP\_X\_FORWARDED\_PROTO", "https")

This setting will not interfere with local development and ensures Django correctly detects secure requests behind Render’s proxy.

**Git**

There’s been a lot of code changes in this chapter so make sure to commit everything with Git.

Remove-Item -Recurse -Force .git

git init

git status

git add .

git commit -m “Chapter 17. Security”

Copy-Item -Recurse -Path "C:\Users\Jean-Marc H\Documents\Django for professionals\Chapter 18. Deployment" -Destination "C:\Users\Jean-Marc H\Documents\Django for professionals\Chapter 18. Deployment - Backup"

If you have any errors make sure to look at your logs with docker-compose logs and compare your code with the official source code on Github

**Render**

Head over to the Render website and sign up for a free account. After verifying your email, Render will redirect you to your dashboard.

To deploy your Django project, push your code to a GitHub repository. Then connect that repo to Render via the Render dashboard.

Once connected, Render will handle the deployment for you. You’ll use your regular terminal (like Git Bash or VS Code's terminal) to push updates to GitHub, and Render will automatically redeploy them.

If you run into any issues, you can visit Render’s documentation or support for help.

Next run your final Git and what is follow:

**GitHub Setup Steps:**

Open an account on Render (✅ already done see above).

Go to GitHub and create a new repository. Name it:

Bookstore-by-Chapter

In the Description, type:

Django for Professionals, structured chapter by chapter for easier understanding and deployment.

Choose Private, uncheck “Initialize this repository with a README.”

Click on Create repository.

On the next screen (GitHub gives push instructions), choose Option 2: …or push an existing repository from the command line.

Copy the 2 lines of code.

Paste into your terminal and press Enter.

✅ Done with GitHub setup.

➡ Now go to Render.com to deploy the project.

**Go to Render and Start Deployment**

Go to https://render.com

Log in or create an account

Click "Deploy your app for free."→ select “Web Service”

On the Connect a Repository screen: Click “GitHub”

Authorize Render to access your GitHub (except if you already did)

Choose “Only select repositories”

Select the repository: Bookstore-by-Chapter

Click “Install”

You will be redirected to “Configure and deploy your new Web Service”

⚙️ Configure and Deploy Your New Web Service

Name: bookstore (or anything you like; lowercase, no spaces)

Region: Keep the default (usually Oregon (US West))

Branch

Select branch: main

Root Directory

Leave it blank (since your manage.py is in the root)

Click on “Deploy Web Service”

A window pop up and ask for your Debit/Credit card.

Render requests a credit/debit card even for the free tier, primarily to confirm your identity and prevent spam. You won’t be charged unless you move to a paid plan or use billable features.

✅ Here’s What to Do Next:

Step Action – **IT’S UP TO YOU TO GIVE DEBIT/CREDIT CARDS INFORMATION YOU ALREADY LEARN ALL STEPS TO DEPLOY.**

1. Add your card Go ahead and input your card details so Render can verify your account.

2. Confirm it’s free In the plan details, ensure the Free instance is selected ($0/month).

3. Continue deployment After card verification, click the "Create Web Service" button to proceed.

No billing will occur unless you purposely switch to a paid instance type. After completing this step, your app will begin building and deploying automatically. When deployment starts, you’ll have access to live logs and a deployment URL.

**This step is not on this book. Step I have to do to get all subfolders on GitHb.**

1. On the main folder create a new subfolder with the exact name you have on GitHb repository

2. Drag all subfoders inside of the new subfolder.

3. Make sure the path is correct. For example:

The one you use to study or work on

PS C:\Users\Jean-Marc H\Documents\Django for professionals\Chapter 18. Deployment>

Shoud become:  
PS C:\Users\Jean-Marc H\Documents\Django for professionals\Bookstore-by-Chapter\Chapter 18. Deployment>

**Here the procedure to transfer all subfolders (chapter) to GitHub:**

Make sure your main folder on your laptop/desktop has the same name then the repository you create on your GitHub account.

1. Open the chapter you want to transfer.

2. Run “cd ..” to get the right patch.

3. If you created an .gitignore make sure “\*docx” is not inside, if it is delete it, leave only “~\*” and “\*.tmp” ignored.

4. Run:

git add "Chapter X. *title of the chapter*"

git commit -m "Add Chapter X. *title of the chapter* "

git push origin main

5. close the chapter and open the next one and do the dame pocedure.

The end.