Chapter 3: Pages App

djangoforbeginners.com/pages-app

Build and deploy a Django app using templates and class-based views

In this chapter we will build, test, and deploy a *Pages* app containing a homepage and about page. We still aren't touching the database yet–that comes in the next chapter–but we'll learn about class-based views and templates which are the building blocks for the more complex web applications built later on in the book.

Initial Set Up

As in Chapter 2: Hello World App, our initial set up involves the following steps:

- make a new directory for our code called `pages` and navigate into it
- create a new virtual environment called `.venv` and activate it
- install Django
- create a new Django project called `django_project`
- create a new app called `pages`

On the command line make sure you're not working in an existing virtual environment. If there is text before the command line prompt–either `>` on Windows or `%` on macOS–then you are! Make sure to type `deactivate` to leave it.

Within a new command line shell start by typing the following:

```
# Windows
> cd onedrive\desktop\code
> mkdir pages
> cd pages
> python -m venv .venv
> .venv\Scripts\Activate.ps1
(.venv) > python -m pip install django~=4.0.0
(.venv) > django-admin startproject django_project .
(.venv) > python manage.py startapp pages

# macOS
% cd ~/desktop/code
% mkdir pages
% cd pages
% python3 -m venv .venv
% source .venv/bin/activate
(.venv) % python3 -m pip install django~=4.0.0
(.venv) % django-admin startproject django_project .
(.venv) % python3 manage.py startapp pages
```

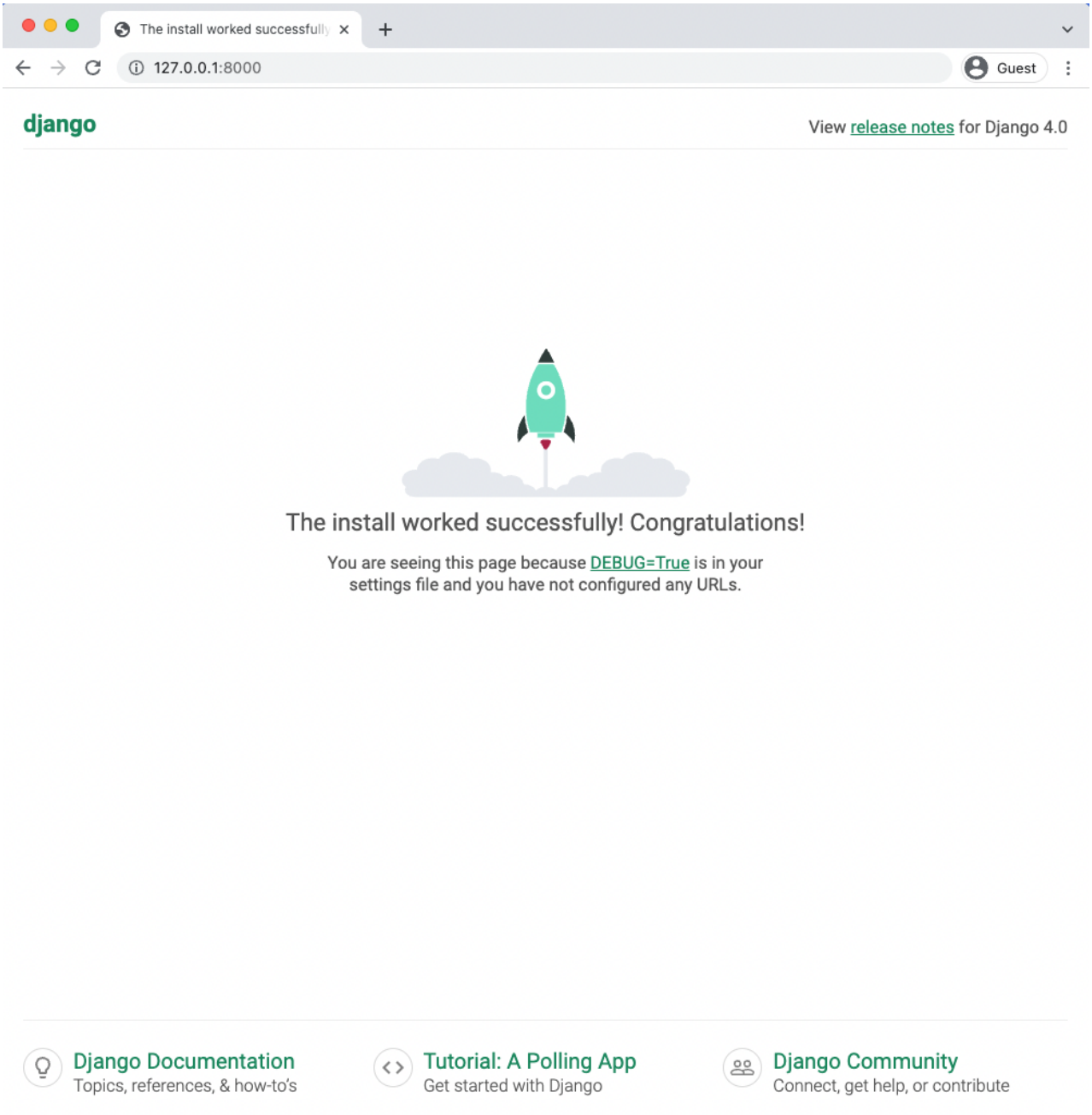
Remember that even though we added a new app, Django will not recognize it until it is added to the `INSTALLED_APPS` setting within `django_project/settings.py`. Open your text editor and add it to the bottom now:

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "pages.apps.PagesConfig",  # new
]
```

Migrate the database with `migrate` and start the local web server with `runserver`.

```
(.venv) > python manage.py migrate
(.venv) > python manage.py runserver
```

And then navigate to `http://127.0.0.1:8000/`.



Templates

Every web framework needs a convenient way to generate HTML files and in Django the approach is to use *templates*: individual HTML files that can be linked together and also include basic logic.

Recall that in the previous chapter our "Hello, World" site had the phrase hardcoded into a `views.py` file. That technically works but doesn't scale well! A better approach is to link a view to a template, thereby separating the information contained in each.

In this chapter we'll learn how to use templates to more easily create our desired homepage and about page. And in future chapters, the use of templates will support building websites that can support hundreds, thousands, or even millions of webpages with a minimal amount of code.

The first consideration is where to place templates within the structure of a Django project. There are two options. By default, Django's template loader will look within each app for related templates. However the structure is somewhat confusing: each app needs a new `templates` directory, another directory with the same name as the app, and then the template file.

Therefore, in our `pages` app, Django would expect the following layout:

```
└── pages
    ├── templates
        ├── pages
            ├── home.html
```

This means we would need to create a new `templates` directory, a new directory with the name of the app, `pages`, and finally our template itself which is `home.html`.

Why this seemingly repetitive approach? The short answer is that the Django template loader wants to be really sure it finds the correct template! What happens if there are `home.html` files within two separate apps? This structure makes sure there are no such conflicts.

There is, however, another approach which is to instead create a single project-level `templates` directory and place *all* templates within there. By making a small tweak to our `django_project/settings.py` file we can tell Django to *also* look in this directory for templates. That is the approach we'll use.

First, quit the running server with the `Control+c` command. Then create a directory called `templates`.

```
(.venv) > mkdir templates
```

Next we need to update `django_project/settings.py` to tell Django the location of our new `templates` directory. This is a one-line change to the setting `"DIRS"` under `TEMPLATES`.

```
# django_project/settings.py
TEMPLATES = [
    {
        ...
        "DIRS": [BASE_DIR / "templates"],  # new
        ...
    },
]
```

Within the `templates` directory create a new file called `home.html`. You can do this within your text editor: in Visual Studio Code go to the top left of your screen, click on "File" and then "New File." Make sure to name and save the file in the correct location.

The `home.html` file will have a simple headline for now.

```
<!-- templates/home.html -->
<h1>Homepage</h1>
```

Ok, our template is complete! The next step is to configure our URL and view files.

Class-Based Views

Early versions of Django only shipped with function-based views, but developers soon found themselves repeating the same patterns over and over again. Write a view that lists all objects in a model. Write a view that displays only one detailed item from a model. And so on.

Function-based generic views were introduced to abstract these patterns and streamline development of common patterns. However, there was no easy way to extend or customize these views. As a result, Django introduced class-based generic views that make it easy to use and also extend views covering common use cases.

Classes are a fundamental part of Python but a thorough discussion of them is beyond the scope of this book. If you need an introduction or refresher, I suggest reviewing the official Python docs which have an excellent tutorial on classes and their usage.

In our view we will use the built-in TemplateView to display our template. Update the `pages/views.py` file.

```
# pages/views.py
from django.views.generic import TemplateView


class HomePageView(TemplateView):
    template_name = "home.html"
```

Note that we've capitalized our view, `HomePageView`, since it's now a Python class. Classes, unlike functions, should always be capitalized. The `TemplateView` already contains all the logic needed to display our template, we just need to specify the template's name.

URLs

The last step is to update our URLs. Recall from Chapter 2 that we need to make updates in two locations. First, we update the `django_project/urls.py` file to point at our `pages` app and then within `pages` we match views to URL routes.

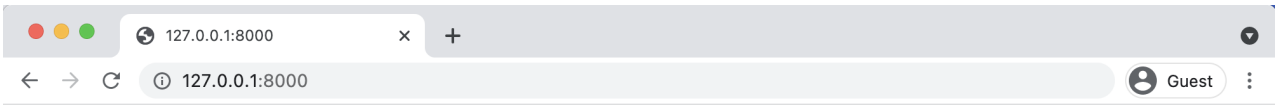
Let's start with the `django_project/urls.py` file.

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include  # new

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

The code here should look familiar at this point. We add `include` on the second line to point the existing URL to the `pages` app. Next create a file called `pages/urls.py` file and add the code below. This pattern is almost identical to what we did in Chapter 2 with one major difference: when using Class-Based Views, you always add `as_view()` at the end of the view name.

```
# pages/urls.py
from django.urls import path
from .views import HomePageView

urlpatterns = [
    path("", HomePageView.as_view(), name="home"),
]
```

And we're done! Start up the local web server with the command `python manage.py runserver` and navigate to `http://127.0.0.1:8000/` to see our new homepage.



Homepage

About Page

The process for adding an about page is very similar to what we just did. We'll create a new template file, a new view, and a new url route. Start by making a new template file called `templates/about.html` and populate it with a short HTML headline.

```
<!-- templates/about.html -->
<h1>About page</h1>
```

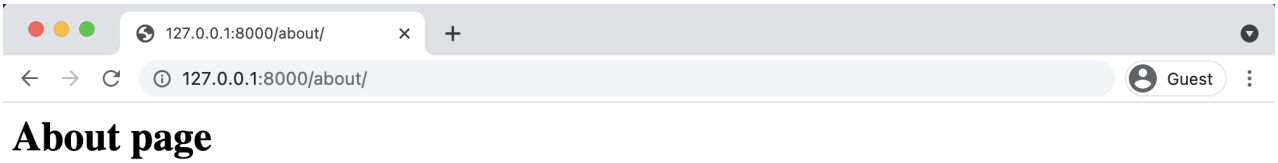
Then add a new view for the page.

```
# pages/views.py
from django.views.generic import TemplateView


class HomePageView(TemplateView):
    template_name = "home.html"


class AboutPageView(TemplateView):  # new
    template_name = "about.html"
```

And finally import the view name and connect it to a URL at `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"),
]
```

Start up the web server with `runserver` and navigate to `http://127.0.0.1:8000/about`. The new About page is visible.



Extending Templates

The real power of templates is their ability to be extended. If you think about most websites, there is content that is repeated on every page (header, footer, etc). Wouldn't it be nice if we, as developers, could have *one canonical place* for our header code that would be inherited by all other templates?

Well we can! Let's create a `base.html` file containing a header with links to our two pages. We could name this file anything but using `base.html` is a common convention. In your text editor make this new file called `templates/base.html`.

Django has a minimal templating language for adding links and basic logic in our templates. You can see the full list of built-in template tags here in the official docs. Template tags take the form of `{% something %}` where the "something" is the template tag itself. You can even create your own custom template tags, though we won't do that in this book.

To add URL links in our project we can use the built-in url template tag which takes the URL pattern name as an argument. Remember how we added optional URL names to our two routes in `pages/urls.py`? This is why. The `url` tag uses these names to

automatically create links for us.

The URL route for our homepage is called `home`. To configure a link to it we use the following syntax: `{% url 'home' %}`.

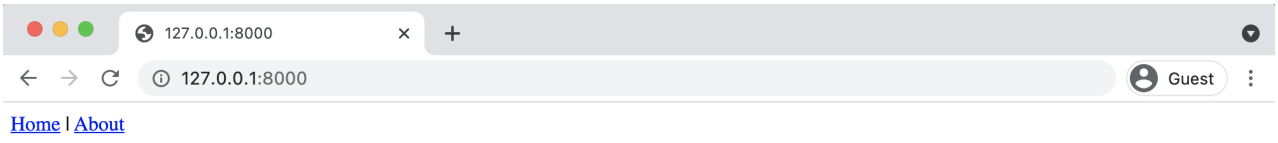
```
<!-- templates/base.html -->
<header>
  <a href="{% url 'home' %}">Home</a> |
  <a href="{% url 'about' %}">About</a>
</header>

{% block content %} {% endblock content %}
```

At the bottom we've added a `block` tag called `content`. Blocks can be overwritten by child templates via inheritance. While it's optional to name our closing `endblock` –you can just write `{% endblock %}` if you prefer–doing so helps with readability, especially in larger template files.

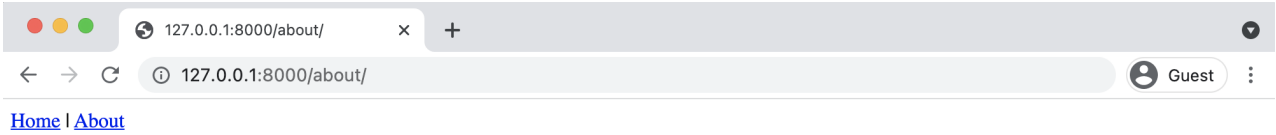
Let's update our `home.html` and `about.html` files to extend the `base.html` template. That means we can reuse the same code from one template in another template. The Django templating language comes with an extends method that we can use for this.

```
<!-- templates/home.html -->
{% extends "base.html" %}

{% block content %}
<h1>Homepage</h1>
{% endblock content %}

<!-- templates/about.html -->
{% extends "base.html" %}

{% block content %}
<h1>About page</h1>
{% endblock content %}
```

Now if you start up the server with `python manage.py runserver` and open up our webpages again at `http://127.0.0.1:8000/` the header is included.



And it is also present on the about page at `http://127.0.0.1:8000/about`.



About page

There's *a lot* more we can do with templates and in practice you'll typically create a `base.html` file and then add additional templates on top of it in a robust Django project. We'll do this later on in the book.

Tests

It's important to add automated tests and run them whenever a codebase changes. Tests require a small amount of upfront time to write but more than pay off later on. In the words of Django co-creator Jacob Kaplan-Moss, "Code without tests is broken as designed."

Testing can be divided into two main categories: unit and integration. *Unit tests* check a piece of functionality in isolation, while *Integration tests* check multiple pieces linked together. Unit tests run faster and are easier to maintain since they focus on only a small piece of code. Integration tests are slower and harder to maintain since a failure doesn't point you in the specific direction of the cause. Most developers focus on writing many unit tests and a small amount of integration tests.

The Python standard library contains a built-in testing framework called unittest that uses TestCase instances and a long list of assert methods to check for and report failures.

Django's own testing framework provides several extensions on top of Python's `unittest.TestCase` base class. These include a test client for making dummy Web browser requests, a number of Django-specific additional assertions, and four test case classes: SimpleTestCase, TestCase, TransactionTestCase, and LiveServerTestCase.

Generally speaking, `SimpleTestCase` is used when a database is not necessary while `TestCase` is used when you do want to test the database. `TransactionTestCase` is useful if you need to directly test database transactions while `LiveServerTestCase` launches a live server thread useful for testing with browser-based tools like Selenium.

One quick note before we proceed: you may notice that the *naming* of methods in `unittest` and `django.test` are written in `camelCase` rather than the more Pythonic `snake_case` pattern. The reason is that `unittest` is based on the `jUnit` testing framework from Java, which does use camelCase, so when `unittest` was added to Python it came along with camelCase naming.

If you look within our `pages` app, Django already provided a `tests.py` file we can use. Since no database is involved yet in our project we will import `SimpleTestCase` at the top of the file. For our first tests we'll check that the two URLs for our website, the

homepage and about page, both return HTTP status codes of 200, the standard response for a successful HTTP request.

```
# pages/tests.py
from django.test import SimpleTestCase


class HomepageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)


class AboutpageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/about/")
        self.assertEqual(response.status_code, 200)
```

To run the tests quit the server with `Control+c` and type `python manage.py test` on the command line to run the tests.

```
(.venv) > python manage.py test
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.006s

OK
```

If you see an error such as `AssertionError: 301 != 200` it's likely you forgot to add the trailing slash to `"/about"` above. The web browser knows to automatically add a slash if it's not provided, but that causes a 301 redirect, not a 200 success response!

What else can we test? How about the URL name for each page? Recall that in `pages/urls.py` we added the name of `"home"` for the homepage path and `"about"` for the about page. We can use the very handy Django utility function reverse to check both. Make sure to import `reverse` at the top of the file and add a new unit test for each below.

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse  # new


class HomepageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_url_available_by_name(self):  # new
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)


class AboutpageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/about/")
        self.assertEqual(response.status_code, 200)

    def test_url_available_by_name(self):  # new
        response = self.client.get(reverse("about"))
        self.assertEqual(response.status_code, 200)
```

Run the tests again to confirm that they work correctly.

```
(.venv) > python manage.py test
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 4 tests in 0.007s

OK
```

We have tested our URL locations and URL names so far but not our templates. Let's make sure that the correct templates– `home.html` and `about.html` –are used on each page and that they display the expected content of "<h1>Homepage</h1>" and "<h1>About page</h1>" respectively.

We can use assertTemplateUsed and assertContains to achieve this.

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse


class HomepageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_url_available_by_name(self):
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)

    def test_template_name_correct(self):  # new
        response = self.client.get(reverse("home"))
        self.assertTemplateUsed(response, "home.html")

    def test_template_content(self):  # new
        response = self.client.get(reverse("home"))
        self.assertContains(response, "<h1>Homepage</h1>")


class AboutpageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/about/")
        self.assertEqual(response.status_code, 200)

    def test_url_available_by_name(self):
        response = self.client.get(reverse("about"))
        self.assertEqual(response.status_code, 200)

    def test_template_name_correct(self):  # new
        response = self.client.get(reverse("about"))
        self.assertTemplateUsed(response, "about.html")

    def test_template_content(self):  # new
        response = self.client.get(reverse("about"))
        self.assertContains(response, "<h1>About page</h1>")
```

Run the tests one last time to check our new work. Everything should pass.

```
(.venv) > python manage.py test
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 8 tests in 0.009s

OK
```

Experienced programmers may look at our testing code and note that there is a lot of repetition. For example, we are setting the `response` each time for all eight tests. Generally it is a good idea to abide by the concept of DRY (Don't Repeat Yourself) coding, but unit tests work best when they are self contained and extremely verbose. As a test suite expands, for performance reasons it might make more sense to combine multiple assertions into a smaller number of test but that is an advanced–and often subjective–topic beyond the scope of this book.

We'll do much more with testing in the future, especially once we start working with databases. For now, it's important to see how easy and important it is to add tests each and every time we add new functionality to our Django project.

Git and GitHub

It's time to track our changes with Git and push them up to GitHub. We'll start by initializing our directory and checking the status of our changes.

```
(.venv) > git init
(.venv) > git status
```

Use `git status` to see all our code changes. Notice that the `.venv` directory with our virtual environment is included? We don't want that. In your text editor create a new hidden file, `.gitignore`, so we can specify what Git will *not* track.

```
.venv/
```

Run `git status` again to confirm these two files are being ignored, then use `git add -A` to add the intended files/directories, and finally add an initial commit message.

```
(.venv) > git status
(.venv) > git add -A
(.venv) > git commit -m "initial commit"
```

Over on GitHub create a new repo called `pages` and make sure to select the "Private" radio button. Then click on the "Create repository" button.

On the next page, scroll down to where it says "...or push an existing repository from the command line." Copy and paste the two commands there into your terminal.

It should look like the below albeit instead of `wsvincent` as the username it will be your GitHub username.

```
(.venv) > git remote add origin https://github.com/wsvincent/pages.git
(.venv) > git push -u origin main
```

Local vs Production

To make our site available on the Internet where everyone can see it, we need to deploy our code to an external server and database. This is called putting our code into *production*. Local code lives only on our computer; production code lives on an external server available to everyone.

The `startproject` command creates a new project configured for local development via the file `django_project/settings.py`. This ease-of-use means when it does come time to push the project into production, a number of settings have to be changed.

One of these is the web server. Django comes with its own basic server, suitable for local usage, but it *is not* suitable for production. There are two options available: Gunicorn and uWSGI. Gunicorn is the simpler to configure and more than adequate for our projects so

that will be what we use.

For our hosting provider we will use Heroku because it is free for small projects, widely-used, and has a relatively straightforward deployment process.

Heroku

You can sign up for a free Heroku account on their website. Fill in the registration form and wait for an email with a link to confirm your account. This will take you to the password setup page. Once configured, you will be directed to the dashboard section of the site. Heroku now requires enrolling in multi-factor authentication (MFA), too, which can be done with SalesForce or a tool like Google Authenticator.

Heroku may also ask about adding a credit card for account verification. This is optional but provides extra benefits even if you use the free tier such as adding a custom domain and adding extra add-ons for additional functionality.

Now that you are signed up, it is time to install Heroku's *Command Line Interface (CLI)* so we can deploy from the command line. Currently, we are operating within a virtual environment for our *Pages* project but we want Heroku available globally, that is everywhere on our machine. An easy way to do so is open up a new command line tab–`Command+t` on a Mac, `Control+t` on Windows–which is not operating in a virtual environment. Anything installed here will be global.

On Windows, see the Heroku CLI page to correctly install either the 32-bit or 64-bit version. On a Mac, the package manager Homebrew is used for installation. If not already on your machine, install Homebrew by copy and pasting the long command on the Homebrew website into your command line and hitting `Return`. It will look something like this:

```
% /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/\
  install/HEAD/install.sh)"
```

Next install the Heroku CLI by copy and pasting the following into your command line and hitting `Return`.

```
% brew tap heroku/brew && brew install heroku
```

If you are on a new M1 chip Apple computer you might receive an error with something like `Bad CPU type in executable`. Installing Rosetta 2 will solve the issue.

Once installation is complete you can close the new command line tab and return to the initial tab with the `pages` virtual environment active.

Type the command `heroku login` and use the email and password for Heroku you just set.

```
(.venv) > heroku login
Enter your Heroku credentials:
Email: will@wsvincent.com
Password: ********************************
Logged in as will@wsvincent.com
```

You are logged in and ready to go!

Deployment Checklist

Deployment typically requires a number of discrete steps. It is common to have a "checklist" for these since there quickly become too many to remember. At this stage, we are intentionally keeping things basic however this list will grow in future projects as we add additional security and performance features.

Here is the current deployment checklist:

- install `Gunicorn`
- create a `requirements.txt` file
- update `ALLOWED_HOSTS` in `django_project/settings.py`
- create a `Procfile`
- create a `runtime.txt` file

The first item is to install Gunicorn, a production-ready web server for our project. Recall that we have been using Django's own lightweight server for development purposes but it is not suitable for a live website. Gunicorn can be installed using Pip.

```
(.venv) > python -m pip install gunicorn==20.1.0
```

Step two is to create a `requirements.txt` file containing all the specific Python dependencies in our project. That is, every Python package currently installed in our virtual environment. This is necessary in case we–or a team member–want to recreate the repository from scratch in the future. It also helps Heroku recognize that this is a Python project, which simplifies the deployment steps.

To create this file, we will redirect the output of the `pip freeze` command to a new file called `requirements.txt`.

```
(.venv) > python -m pip freeze > requirements.txt
```

If you look at the contents of this file it contains `Django`, `gunicorn`, `asgiref` and `sqlparse`. That's because when we installed Django we *also* end up installing a number of other Python modules it depends upon. You do not need to focus too much on the contents of this file, just remember that whenever you install a new Python package in your virtual environment you should update the `requirements.txt` file to reflect it. There are more complicated tools that automate this process but since we are using `venv` and keeping things as simple as possible, it must be done manually.

The third step is to look within `django_project/settings.py` for the ALLOWED_HOSTS setting, which represents that host/domain names our Django site can serve. This is a security measure to prevent HTTP Host header attacks. For now, we'll use the wildcard asterisk, `*` so that *all* domains are acceptable. Later in the book we'll see how to explicitly list the domains that should be allowed which is much more secure.

```
# django_project/settings.py
ALLOWED_HOSTS = ["*"]
```

The fourth step is to create a new `Procfile` in the base directory next to the `manage.py` file. The `Procfile` is specific to Heroku and provides instructions on how to run the application in their stack. We're indicating that for the `web` function use `gunicorn` as the server, the WSGI config file located at `django_project.wsgi`, and finally the flag `--log-file -` makes any logging messages visible to us.

```
web: gunicorn django_project.wsgi --log-file -
```

The final step is to specify which Python version should run on Heroku. If not set explicitly this is currently set to the `python-3.9.10` runtime but changes over time. Since we are using Python 3.10 we must create a dedicated runtime.txt file to use it. In your text editor, create this new `runtime.txt` file at the project-level meaning it is in the same directory as `manage.py` and the `Procfile`.

To check the current version of Python run `python --version`.

```
# Windows
(.venv) > python --version
Python 3.10.2

# macOS
(.venv) % python3 --version
Python 3.10.2
```

In the base directory, right next to the `Procfile`, create a new file called `runtime.txt` and use the following format. Make sure everything is lowercased!

```
python-3.10.2
```

That's it! Keep in mind we've taken a number of security shortcuts here but the goal is to push our project into production in as few steps as possible. In later chapters we will cover proper security around deployments in depth.

Use `git status` to check our changes, add the new files, and then commit them:

```
(.venv) > git status
(.venv) > git add -A
(.venv) > git commit -m "New updates for Heroku deployment"
```

Finally, push to GitHub so we have an online backup of our code changes.

```
(.venv) > git push -u origin main
```

The last step is to actually deploy with Heroku. If you've ever configured a server yourself in the past, you'll be amazed at how much simpler the process is with a platform-as-a-service provider like Heroku.

Our process will be as follows:

- create a new app on Heroku
- disable the collection of static files (we'll cover this in a later chapter)
- push the code up to Heroku
- start the Heroku server so the app is live
- visit the app on Heroku's provided URL

We can do the first step, creating a new Heroku app, from the command line with `heroku create`. Heroku will create a random name for our app, in my case `fathomless-hamlet-26076`. Your name will be different.

```
(.venv) > heroku create
Creating app... done, ⬢ fathomless-hamlet-26076
https://fathomless-hamlet-26076.herokuapp.com/ |
https://git.heroku.com/fathomless-hamlet-26076.git
```

The `heroku create` command also creates a dedicated remote named `heroku` for our app. To see this, type `git remote -v`.

```
> git remote -v
heroku  https://git.heroku.com/fathomless-hamlet-26076.git (fetch)
heroku  https://git.heroku.com/fathomless-hamlet-26076.git (push)
```

This new remote means that as long as we include `heroku` in a command we have the ability to both push and fetch code from Heroku now.

At this point, we only need one set of additional Heroku configurations, which is to tell Heroku to ignore static files like CSS and JavaScript. By default, Django will try to optimize these for us which causes issues. We'll cover this in later chapters so for now just run the following command:

```
(.venv) > heroku config:set DISABLE_COLLECTSTATIC=1
```

Now we can push our code up to Heroku:

```
(.venv) > git push heroku main
```

If we had just typed `git push origin main` the code would have been pushed to GitHub, not Heroku. Adding `heroku` to the command sends the code to Heroku. Git itself can be quite confusing initially to understand!

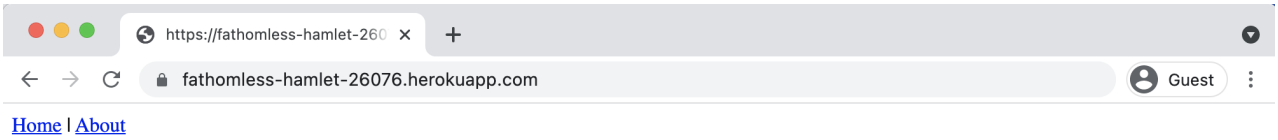
The last step is to make our Heroku app live. As websites grow in traffic they need additional Heroku services but for our basic example we can use the lowest level, `web=1`, which also happens to be free! Type the following command:

```
(.venv) > heroku ps:scale web=1
```

We're done! The last step is to confirm our app is live and online. If you use the command `heroku open` your web browser will open a new tab with the URL of your app:

```
(.venv) > heroku open
```

Mine is at `https://fathomless-hamlet-26076.herokuapp.com/`.



Homepage

You do not have to log out or exit from your Heroku app. It will continue running at this free tier on its own, though you should type `deactivate` to leave the local virtual environment and be ready for the next chapter.

Conclusion

Congratulations on building and deploying your second Django project! This time we used templates, class-based views, explored URLs more fully, added basic tests, and used Heroku. If you feel overwhelmed by the deployment process, don't worry. Deployment *is* hard even with a tool like Heroku. The good news is the steps required are the same for most projects so you can refer to a deployment checklist to guide you each time you launch a new project.

The full source code for this chapter is available on GitHub if you need a reference. In the next chapter we'll move on to our first database-backed project, a *Message Board* website, and see where Django really shines.

The full source code for this chapter is available on GitHub if you need a reference.

Continue on to Chapter 4: Message Board where we'll build and deploy a more complex Django application using templates and class-based views.