Chapter 2: Hello World app

djangoforbeginners.com/hello-world

Build a "Hello, World" Django application

In this chapter we will build a Django project that simply says "Hello, World" on the homepage. This is the traditional way to start a new programming language or framework. We'll also work with Git for the first time and deploy our code to GitHub.

If you become stuck at any point, complete source code for this and all future chapters is available online on the official GitHub repo for the book.

Initial Set Up

To begin, open up a new command line shell or use the built-in terminal on VS Code. For the latter click on "Terminal" at the top and then "New Terminal" to bring it up on the bottom of the screen.

Make sure you are not in an existing virtual environment by checking there is nothing in parentheses before your command line prompt. You can even type `deactivate` to be completely sure. Then navigate to the `code` directory on your Desktop and create a `helloworld` directory with the following commands.

```
# Windows
> cd onedrive\desktop\code
> mkdir helloworld
> cd helloworld

# macOS
% cd ~/desktop/code
% mkdir helloworld
% cd helloworld
```

Create a new virtual environment called `.venv`, activate it, and install Django with Pip as we did in the previous chapter.

```
# Windows
> python -m venv .venv
> .venv\Scripts\Activate.ps1
(.venv) > python -m pip install django~=4.0.0

# macOS
% python3 -m venv .venv
% source .venv/bin/activate
(.venv) % python3 -m pip install django~=4.0.0
```

Now we'll use the Django `startproject` command to make a new project called `django_project`. Don't forget to include the period (`.`) at the end of the command so that it is installed in our current directory.

```
(.venv) > django-admin startproject django_project .
```

Let's pause for a moment to examine the default project structure Django has provided for us. You examine this visually if you like by opening the new directory with your mouse on the Desktop. The `.venv` directory may not be initially visible because it is "hidden" but nonetheless still there.

```
├── django_project
|   ├── __init__.py
|   ├── asgi.py
|   ├── settings.py
|   ├── urls.py
|   └── wsgi.py
├── manage.py
└── .venv/
```

The `.venv` directory was created with our virtual environment but Django has added a `django_project` directory and a `manage.py` file. Within `django_project` are five new files:

- `__init__.py` indicates that the files in the folder are part of a Python package. Without this file, we cannot import files from another directory which we will be doing a lot of in Django!
- `asgi.py` allows for an optional Asynchronous Server Gateway Interface to be run
- `settings.py` controls our Django project's overall settings
- `urls.py` tells Django which pages to build in response to a browser or URL request
- `wsgi.py` stands for Web Server Gateway Interface which helps Django serve our eventual web pages.

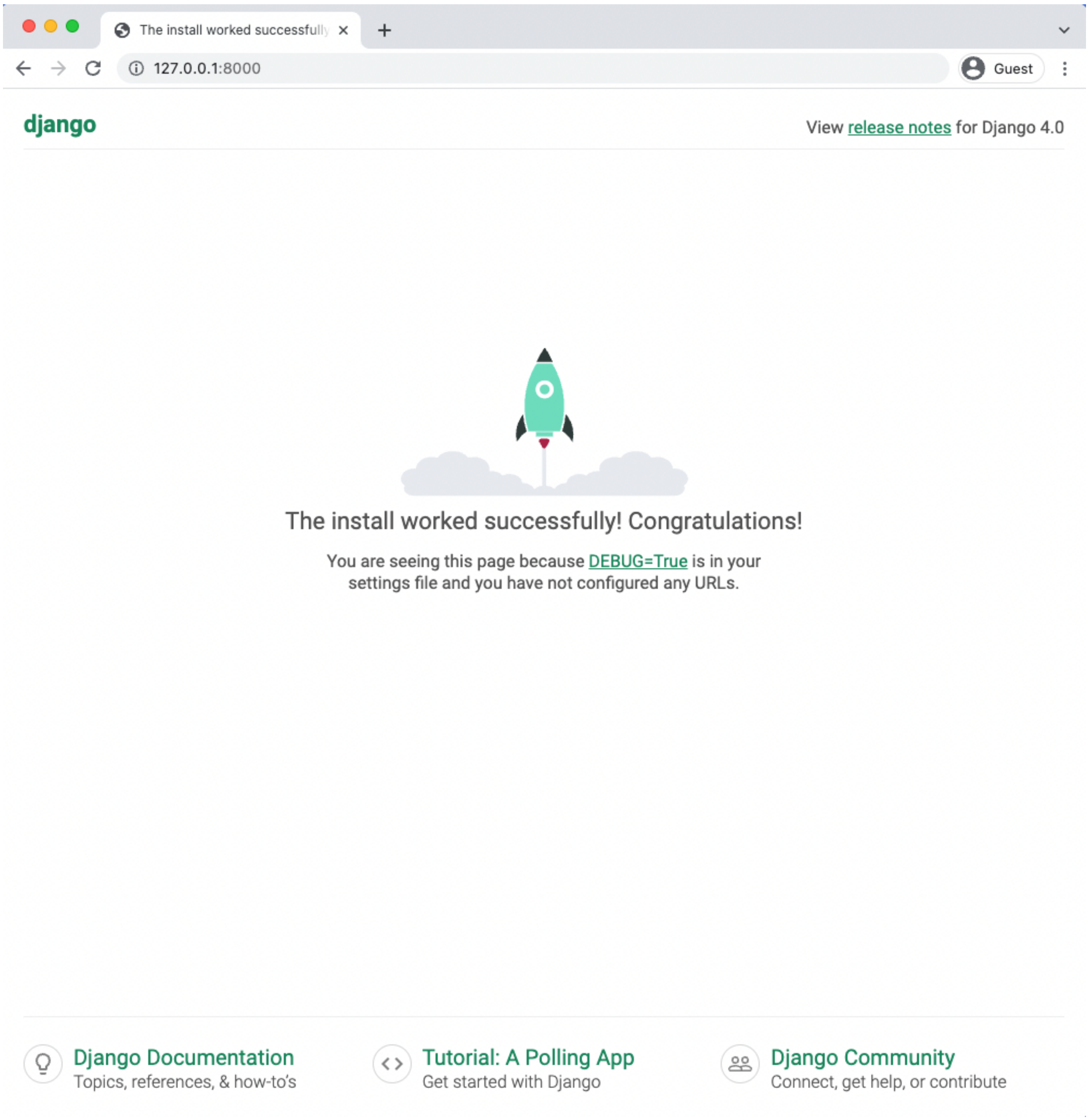
The `manage.py` file is not part of `django_project` but is used to execute various Django commands such as running the local web server or creating a new app.

Let's try out our new project by using Django's lightweight built-in web server for local development purposes. The command we'll use is `runserver` which is located in `manage.py`.

```
# Windows
(.venv) > python manage.py runserver

# macOS
(.venv) % python3 manage.py runserver
```

If you visit `http://127.0.0.1:8000/` you should see the following image:


The install worked successfully
127.0.0.1:8000
Guest
django
View release notes for Django 4.0

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Django Documentation
Topics, references, & how-to's
Tutorial: A Polling App
Get started with Django
Django Community
Connect, get help, or contribute

Note that the full command line output will contain additional information including a warning about `18 unapplied migrations`. Technically, this warning doesn't matter at this point. Django is complaining that we have not yet "migrated" our initial database. Since we won't actually use a database in this chapter the warning won't affect the end result.

However, since warnings are still annoying to see, we can remove it by first stopping the local server with the command `Control+c` and then running `python manage.py migrate`.

```
# Windows
> python manage.py migrate

# macOS
% python3 manage.py migrate

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
```

What Django has done here is create a SQLite database and migrated its built-in apps provided for us. This is represented by the new file `db.sqlite3` in our directly.

```
├── django_project
│   ├── __init__.py
|   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── db.sqlite3  # new
├── manage.py
└── .venv/
```

For now, if you execute `python manage.py runserver` again you should no longer see any warnings.

HTTP Request/Response Cycle

HTTP (Hypertext Transfer Protocol) was initially created by Tim Berners-Lee in 1989 and is the foundation of the World Wide Web. A network protocol is a set of rules for formatting and processing data. It is like a common language for computers that allows them to communicate with one another even if they are located on opposite sides of the earth and have vastly different hardware and software.

HTTP is a request-response protocol that works in a client-server computing model. Every time you visit a webpage an initial “request” is sent by the “client” (ie your computer) and a “response” is sent back by a “server.” The client doesn’t have to be a computer though, it could also be a mobile phone or any internet-connected device. But the process is the same: an HTTP request is sent to a URL by a client and a server sends back an HTTP response.

Ultimately, all a web framework like Django does is accept HTTP requests to a given URL and return a HTTP response containing the information necessary to render a webpage. That's it. Generally this process involves identifying the proper URL, connecting to a database, adding some logic, applying style with HTML/CSS/JavaScript/static assets, and then return the HTTP response. That's it.

The abstract flow looks something like this:

```
HTTP Request -> URL -> Django combines database, logic, styling -> HTTP Response
```

Model-View-Controller vs Model-View-Template

Over time the **Model-View-Controller (MVC)** pattern has emerged as a popular way to *internally* separate the data, logic, and display of an application into separate components. This makes it easier for a developer to reason about the code. The MVC pattern is widely used among web frameworks including Ruby on Rails, Spring (Java), Laravel (PHP), ASP.NET (C#) and many others.

In the traditional MVC pattern there are three major components:

- Model: Manages data and core business logic
- View: Renders data from the model in a particular format
- Controller: Accepts user input and performs application-specific logic

Django only loosely follows the traditional MVC approach with its own version often called **Model-View-Template (MVT)**. This can be initially confusing to developers with previous web framework experience. In reality, Django's approach is really it is a 4-part pattern that also incorporates URL Configuration so something like *MVTU* would be a more accurate description.

The Django MVT pattern is as follows:

- Model: Manages data and core business logic
- View: Describes *which* data is sent to the user but not its presentation
- Template: Presents the data as HTML with optional CSS, JavaScript, and Static Assets
- URL Configuration: Regular-expression components configured to a View

This interaction is fundamental to Django yet **very confusing** to newcomers so let's map out the order of a given HTTP request/response cycle. When you type in a URL, such as `https://djangoproject.com`, the first thing that happens within our Django project is a URL pattern (contained in `urls.py`) is found that matches it. The URL pattern is linked to a single view (contained in `views.py`) which combines the data from the model (stored in `models.py`) and the styling from a template (any file ending in `.html`). The view then returns a HTTP response to the user.

The complete Django flow looks like this:

```
HTTP Request -> URL -> View -> Model and Template -> HTTP Response
```

If you are brand new to web development the distinction between MVC and MVT will not matter much. This book demonstrates the Django way of doing things so there won't be confusion. However if you are a web developer with previous MVC experience, it can take a little while to shift your thinking to the "Django way" of doing things which is more loosely coupled and allows for easier modifications.

Create An App

Django uses the concept of projects and apps to keep code clean and readable. A single top-level Django project can contain multiple apps. Each app controls an isolated piece of functionality. For example, an e-commerce site might have one app for user authentication, another app for payments, and a third app to power item listing details. That's three distinct apps that all live within one top-level project. How and when you split functionality into apps is somewhat subjective, but in general, each app should have a clear function.

To add a new app go to the command line and quit the running server with `Control+c`. Then use the `startapp` command followed by the name of our app which will be `pages`.

```
# Windows
(.venv) > python manage.py startapp pages

# macOS
(.venv) % python3 manage.py startapp pages
```

If you look visually at the `helloworld` directory Django has created within it a new `pages` directory containing the following files:

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

Let's review what each new `pages` app file does:

- `admin.py` is a configuration file for the built-in Django Admin app
- `apps.py` is a configuration file for the app itself

- `migrations/` keeps track of any changes to our `models.py` file so it stays in sync with our database
- `models.py` is where we define our database models which Django automatically translates into database tables
- `tests.py` is for app-specific tests
- `views.py` is where we handle the request/response logic for our web app

Notice that the model, view, and url from the MVT pattern are present from the beginning. The only thing missing is a template which we'll add shortly.

Even though our new app exists within the Django project, Django doesn't "know" about it until we explicitly add it to the `django_project/settings.py` file. In your text editor open the file up and scroll down to `INSTALLED_APPS` where you'll see six built-in Django apps already there. Add `pages.apps.PagesConfig` at the bottom.

```
# 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
]
```

If you have Black installed in your text editor on "save" it will change all the single quotes `''` here to double quotes `""`. That's fine. As noted previously, Django has plans to adopt Black Python formatting fully in the future.

What is `PagesConfig` you might ask? Well, it is the name of the solitary function within the `pages/apps.py` file at this point.

```
# pages/apps.py
from django.apps import AppConfig


class PagesConfig(AppConfig):
    default_auto_field = "django.db.models.BigAutoField"
    name = "pages"
```

Don't worry if you are confused at this point: it takes practice to internalize how Django projects and apps are structured. Over the course of this book we will build many projects and apps and the patterns will soon become familiar.

Hello, World

In Django, four separate files aligning with this MVT pattern are required to power one single dynamic (aka linked to a database) webpage:

- `models.py`
- `views.py`

- `template.html` (any HTML file will do)
- `urls.py`

However, to create a static webpage (not linked to a database) we can hardcode the data into a view so the model is not needed. That's what we'll do here to keep things simple. From Chapter 4 onwards we'll be using the model in all our projects.

The next step is therefore to create our first view. Start by updating the `views.py` file in our `pages` app to look as follows:

```
# pages/views.py
from django.http import HttpResponse


def homePageView(request):
    return HttpResponse("Hello, World!")
```

Basically, we're saying whenever the view function `homePageView` is called, return the text "Hello, World!" More specifically, we've imported the built-in HttpResponse method so we can return a response object to the user. We've created a function called `homePageView` that accepts the `request` object and returns a `response` with the string "Hello, World!"

There are two types of views in Django: function-based views (FBVs) and class-based views (CBVs). Our code in this example is a function-based view: it is relatively simple to implement and explicit. Django originally started with only FBVs but over time added CBVs which allow for much greater code reusability, keeps things DRY (Don't-Repeat-Yourself), and can be extended via mixins. The additional abstraction of CBVs makes them quite powerful and concise, however it also makes them harder to read for Django newcomers.

Because web development quickly becomes repetitive Django also comes with a number of built-in generic class-based views (GCBVs) to handle common use cases such as creating a new object, forms, list views, pagination, and so on. We will be using GCBVs heavily in this book in later chapters.

There are therefore technically three ways to write a view in Django: function-based views (FBVs), class-based views (CBVs), and generic class-based views (GCBVs). This customization is helpful for advanced developers but confusing for newcomers. Many Django developers–including your author–prefer to use GCBVs when possible and revert to CBVs or FBVs when required. By the end of this book you will have used all three and can make up your own mind on which approach you prefer.

Moving along we need to configure our URLs. In your text editor, create a new file called `urls.py` within the `pages` app. Then update it with the following code:

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

urlpatterns = [
    path("", homePageView, name="home"),
]
```

On the top line we import `path` from Django to power our URL pattern and on the next line we import our views. By referring to the `views.py` file as `.views` we are telling Django to look within the current directory for a `views.py` file and import the view `homePageView` from there.

Our URL pattern has three parts:

- a Python regular expression for the empty string `""`
- a reference to the view called `homePageView`
- an optional named URL pattern called `"home"`

In other words, if the user requests the homepage represented by the empty string `""`, Django should use the view called `homePageView`.

We're *almost* done at this point. The last step is to update our `django_project/urls.py` file. It's common to have multiple apps within a single Django project, like `pages` here, and they each need their own dedicated URL path.

```
# 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
]
```

We've imported `include` on the second line next to `path` and then created a new URL pattern for our `pages` app. Now whenever a user visits the homepage, they will first be routed to the `pages` app and then to the `homePageView` view set in the `pages/urls.py` file.

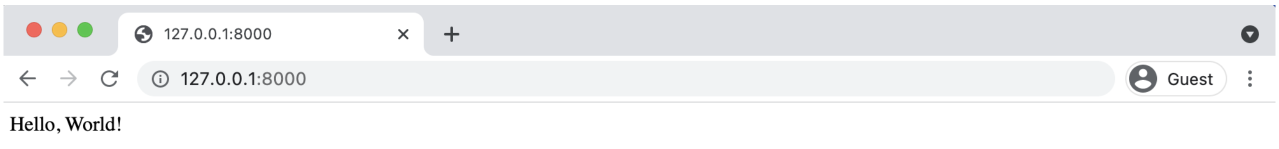
This need for two separate `urls.py` files is often confusing to beginners. Think of the top-level `django_project/urls.py` as the gateway to various url patterns distinct to each app.

We have all the code we need now. To confirm everything works as expected, restart our Django server:

```
# Windows
(.venv) > python manage.py runserver

# macOS
(.venv) % python3 manage.py runserver
```

If you refresh the browser for `http://127.0.0.1:8000/` it now displays the text “Hello, World!”



Hello, World!

Git

In the previous chapter we installed the version control system Git. Let’s use it here. The first step is to initialize (or add) Git to our repository. Make sure you’ve stopped the local server with `Control+c`, then run the command `git init`.

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

If you then type `git status` you’ll see a list of changes since the last Git commit. Since this is our first commit, this list is all of our changes so far.

```
(.venv) > git status
On branch main

No commits yet

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

        .venv
        django_project/
        db.sqlite3
        manage.py
        pages/

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

Note that our virtual environment `.venv` is included which is *not* a best practice. It should be kept out of Git source control since secret information such as API keys and the like are often included in it. The solution is to create a new file called `.gitignore` which tells Git what to ignore. The period at the beginning indicates this is a “hidden” file. The file still exists but it is a way to communicate to developers that the contents are probably meant for configuration and not source control. In this new file, add a single line for `.venv`.

```
.venv/
```

If you run `git status` again you will see that `.venv` is not longer there. It has been “ignored” by Git.

At the same time, we *do* want a record of packages installed in our virtual environment. The current best practice is to create a `requirements.txt` file with this information. The command `pip freeze` will output the contents of your current virtual environment and by using the `>` operator we can do all this in one step: output the contents into a new file called `requirements.txt`. If your server is still running enter `Ctrl+c` and `Enter` to exit before entering this command.

```
(.venv) > pip freeze > requirements.txt
```

A new `requirements.txt` file will appear with all our installed packages and their dependencies. If you look *inside* this file you'll see there are actually four packages even though we have installed only one.

```
asgiref==3.4.1
Django==4.0
sqlparse==0.4.2
```

That's because Django relies on *other* packages for support, too. It is often the case that when you install one Python package you're often installing several others it depends on as well. Since it is difficult to keep track of all the packages a `requirements.txt` file is very important.

We next want to add *all* recent changes by using the command `add -A` and then `commit` the changes along with a message (`-m`) describing what has changed.

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

In professional projects a `.gitignore` file is typically quite lengthy. For efficiency and security reasons, there are often quite a few directories and files that should be removed from source control. We will cover this more thoroughly in Chapter 16 when we do a professional-grade deployment.

GitHub

It's a good habit to create a remote repository of your code for each project. This way you have a backup in case anything happens to your computer and more importantly, it allows for collaboration with other software developers. Popular choices include GitHub, Bitbucket, and GitLab. When you're learning web development, it's best to stick to private rather than public repositories so you don't inadvertently post critical information such as passwords online.

We will use GitHub in this book but all three services offer similar functionality for newcomers. Sign up for a free account on GitHub's homepage and verify your email address. Then navigate to the "Create a new repository" page located at https://github.com/new.

Enter the repository name `hello-world` and click on the radio button next to "Private" rather than "Public." Then click on the button at the bottom for "Create Repository."

Your first repository is now created! However there is no code in it yet. Scroll down on the page to where it says "…or push an existing repository from the command line." That's what we want. Copy the text immediately under this headline and paste it into your command line. Here is what it looks like for me with my GitHub username of `wsvincent`. Your username will be different.

```
> git remote add origin https://github.com/wsvincent/hello-world.git
> git branch -M main
> git push -u origin main
```

The first line syncs the local directory on our computer with the remote repository on the GitHub website. The next line establishes the default branch as `main` which is what we want. And then the third line "pushes" the code up to GitHub's servers.

Assuming everything worked properly, you can now go back to your GitHub webpage and refresh it. Your local code is now hosted online!

SSH Keys

Unfortunately, there is a good chance that the last command yielded an error if you are a new developer and do not have SSH keys already configured.

```
ERROR: Repository not found.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

This cryptic message means we need to configure SSH keys. This is a one-time thing but a bit of a hassle to be honest.

SSH is a protocol used to ensure private connections with a remote server. Think of it as an additional layer of privacy on top of username/password. The process involves generating unique SSH keys and storing them on your computer so only GitHub can access them.

First, check whether you have existing SSH keys. Github has a guide to this that works for Mac, Windows, and Linux. If you *don't* have existing public and private keys, you'll need to generate them. GitHub, again, has a guide on doing this.

Once complete you should be able to execute the `git push -u origin main` command successfully!

It's normal to feel overwhelmed and frustrated if you become stuck with SSH keys. GitHub has a lot of resources to walk you through it but the reality is its very intimidating the first time. If you're truly stuck, continue with the book and come back to SSH Keys and GitHub with a full nights sleep. I can't count the number of times a clear head has helped me process a difficult programming issue.

Assuming success with GitHub, go ahead and exit our virtual environment with the `deactivate` command.

```
(.venv) > deactivate
```

You should no longer see parentheses on your command line, indicating the virtual environment is no longer active.

Conclusion

Congratulations! We've covered a lot of fundamental concepts in this chapter. We built our first Django application and learned about Django's project/app structure. We started to learn about views, urls, and the internal Django web server. And we worked with Git to track our changes and pushed our code into a private repo on GitHub.

If you became stuck at any point make sure to compare your code against the official repo.

Continue on to Chapter 3: Pages App where we'll build and deploy a more complex Django application using templates and class-based views.