18

DEPLOYMENT OF A DJANGO APPLICATION (PART 2 – CONFIGURATION AND CODE DEPLOYMENT)

OVERVIEW

In this chapter, you will begin by creating **PostgreSQL** users and databases. You will then create system users and study the important aspects of file and directory ownership. You will learn how to create a virtual environment on your virtual server, and how to upload your code with an SFTP client. You'll then see how to install the Python requirements for your Django project. You'll configure `systemd` to run **Gunicorn**, then add configuration for **NGINX**. Finally, you'll be able to see your Django application running under production software on your virtual machine.

INTRODUCTION

This chapter continues from *Chapter 17*, *Deployment of a Django Application (Part 1 – Server Setup)*, in which we set up an Ubuntu Linux server and installed the NGINX, Gunicorn, and PostgreSQL software. We will continue along with the process of server setup by uploading our code and configuring our application.

> **NOTE**
>
> We didn't make any code changes in *Chapter 17*, *Deployment of a Django Application (Part 1 – Server Setup)*, so we will use the final *Chapter 16*, *Using a Frontend JavaScript Library with Django*, code as a starting point for this chapter. You can get it from http://packt.live/2LrKOgV.

The first step, though, is to set up a PostgreSQL database to replace the SQLite database we have been using.

When developing and running Django on our local machine we have been using **SQLite**, which is a SQL database that stores all information in a single file. It is fine for use during development when only one person is accessing the site at a time. However, we want our Django apps to be useable by more than one person at once, and that's why we need a dedicated database server. A dedicated server will also allow us to scale in the future: PostgreSQL can store billions of rows and the user can access them quickly. While we don't necessarily need all that capacity for Bookr, it's good to know that our database server won't hold us back when we grow. There are many free and open-source database servers to choose from. We will be using PostgreSQL, chosen for its robustness and speed.

It's also worth noting that in production environments you might encounter other databases such as **MariaDB**, **MSSQL**, or **Oracle**. While they are all configured differently and accessed with different commands, fundamentally they behave the same. They are all standalone database servers that store data in tables, columns, and rows. They can all scale well (to varying degrees) and are accessed using SQL queries.

We saw how the PostgreSQL server was installed in the previous chapter, using **`apt`**. The setup process also started the PostgreSQL server and added a script so it will start when the virtual machine boots up. Soon, we'll be connecting to the server using SSH and creating a user and a database. But before we do that, let's get ourselves versed in some important PostgreSQL commands.

POSTGRESQL COMMANDS

Apart from installing the server, the **postgresql** package can also help us set up PostgreSQL users and databases, thanks to several of the standalone commands that it comes with.

We'll describe the PostgreSQL commands we're going to use (as well as some we won't use but are useful to know), and then in the exercise that follows, we'll use them to set up the database on our virtual machine. These are all commands that are to be run at the command line:

- **createuser**

  This command creates a PostgreSQL user. A newly created PostgreSQL user is different from a system user or Django user. The user won't be able to do anything yet until they have been assigned to a database.

  This command takes one argument, which is the name of the user to create:

```
createuser <username>
```

  We want our new user to have a password to connect to the database, so we should use the **-P** flag to get a password prompt that allows us to enter one:

```
$ createuser username -P
Enter password for new role:
Enter it again:
$
```

- **createdb**

  This command will create a PostgreSQL database. It takes one argument, which is the name of the database to create:

```
createdb <dbname>
```

  By default, the database is owned by the **postgres** database user, but the owner can be set with the **-O** flag:

```
$ createdb databasename -O username
```

- **`psql`**

  This command starts a SQL shell/prompt for the database we want to specify. This is similar to what happens when we use the **`dbshell`** Django management command. When using **`dbshell`**, however, the connection settings will be read from our Django **`settings.py`** file. When using **`psql`**, we should specify options on the command line. The options we will use are:

  `-h <hostname>`: Specify the hostname/address. In our case, we'll use **`127.0.0.1`**.

  `-U <username>`: Specify the PostgreSQL user to connect to the database as.

  `-W`: Prompt for a password when connecting.

  For example:

```
$ psql -U username -W -d databasename -h 127.0.0.1
Password for user username:
psql (10.12 (Ubuntu 10.12-0ubuntu0.18.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384,
bits: 256, compression: off)
Type "help" for help.


databasename=> \q
$
```

  Like the Python shell, we can quit **`psql`** using the *Ctrl* + *D* combination.

- **`dropuser`**

  The **`dropuser`** command is the opposite of the **`createuser`** command. It removes a PostgreSQL user. Like **`createuser`**, it takes one argument, the username to remove (or "drop"):

```
$ dropuser <username>
```

  You cannot drop a user who is the owner of a database. First, drop the database(s) that the user owns, then the user can be dropped:

```
$ sudo -u postgres dropuser username
  dropuser: removal of role "username" failed: ERROR:  role
"username" cannot be dropped because some objects depend on it
  DETAIL:  owner of database databasename
$
```

- **dropdb**

  By now, it's not that hard to guess what the **dropdb** command does. It removes (*drops*) a database. It takes one argument, which is the name of the database to drop:

```
$ drop db <dbname>
```

Those are all the PostgreSQL related commands we'll cover in this book. You will only use the **createuser**, **createdb**, and **psql** commands in the next exercise. The rest are given for reference only.

Before we continue, we need to introduce one more feature of **sudo**: the **-u** flag. This allows you to specify a user to run the command as (other than the default, which is **root**). For example, to run a command as the postgres system user:

```
ben@bookr:~$ sudo -u postgres whoami
[sudo] password for ben:
postgres
ben@bookr:~$
```

All the PostgresSQL commands, except for **psql**, must be run as the **postgres** user. For example, to create a database:

```
sudo -u postgres createdb <databasename>
```

In the next exercise, you will set up a PostgreSQL user and database on your virtual machine.

> **NOTE**
>
> Before *Chapter 17*, *Deployment of a Django Application (Part 1 – Server Setup)*, the code and examples given would work on Windows, macOS, or Linux. In this chapter, the installation and setup guides assume an Ubuntu Linux server. This is because the paths to data and config files differ between operating systems. It is possible to run NGINX and Gunicorn on Windows or macOS too, but you will need to research how to install them yourself.
>
> Throughout this chapter, we will refer to this virtual machine as the "virtual machine." If a command is to be run on it, we will say "Run command … on your virtual machine" or "On your virtual machine, do…", and so on.

EXERCISE 18.01: POSTGRESQL USER AND DATABASE SETUP

In this exercise, you will connect to your virtual machine using SSH, then create a PostgreSQL user and database. You will then validate that the credentials are working correctly by connecting to the database using **`psql`** and executing a simple SQL statement.

> **NOTE**
>
> Although it does not happen often, your connection to the server might be interrupted. Usually, you can reconnect and carry on from the step that you were up to, however you might have to repeat some of the steps to get back to the same state as you were before. For example, if you used the **`sudo`** command to change user, you should run that again. Likewise, if you used **`cd`** to change directory this should be run too. Commands like **`mkdir`**, **`chown`** or **`chmod`** don't need to be rerun, as the directory creation, ownership or mode settings will have already been performed.

1. Start VirtualBox and then start up your virtual machine, if it's not already running. Let it finish booting up – you'll know it's finished because you'll see the login prompt.
2. SSH into your virtual machine using the IP address you found in *Exercise 17.03*, *VirtualBox Networking Configuration*, of *Chapter 17*, *Deployment of a Django Application (Part 1 – Server Setup)* (or **`localhost`**).
3. Once you have connected, create a database user called **`bookr`** using the **`createuser`** command, with the **`-P`** flag to prompt for a password. Run the command using **`sudo`** with the **`-u postgres`** option. Putting it all together, you should run this:

```
sudo -u postgres createuser -P bookr
```

After pressing *Enter*, you might first need to type in your user's password to authenticate **`sudo`** (unless it's been less than 5 minutes since you last used **`sudo`**). The prompts should make it clear when to type in your user password and when to type in the password for the new database user. See the following output as an example:

```
ben@bookr:~$ sudo -u postgres createuser -P bookr
[sudo] password for ben:
Enter password for new role:
```

```
Enter it again:
ben@bookr:~$
```

After creating the user, no output is given, but you can assume that it was a success if no errors appear.

4. Create a new database called **bookr**, using the **createdb** command. It should be owned by the newly created **bookr** user, so use the **-O** flag to specify this. Once again, the command must be run using **sudo** with the **-u postgres** option:

```
sudo -u postgres createdb -O bookr bookr
```

Like the **createuser** command, there will be no output. If you don't see any errors, you can assume the database was created successfully, for example:

```
ben@bookr:~$ sudo -u postgres createdb -O bookr bookr
ben@bookr:~$
```

5. Now test that everything worked by connecting using the **psql** program. This doesn't need to be run as **sudo**, but we do need to specify the user with **-U**, the host with **-h**, and use the **-W** flag to prompt for a password. In the following command, the database name (**bookr**) is specified with a flag:

```
psql -U bookr -h 127.0.0.1 -W bookr
```

Type in the password you set for the user in *step 3*. Then press *Enter*.

After authenticating successfully, you will see some output about the protocol the server is using and how it was built, and then a prompt with the database name (in our case, it is **bookr=>**):

```
ben@bookr:~$ psql -U bookr -h 127.0.0.1 -W bookr
Password for user bookr:
psql (10.12 (Ubuntu 10.12-0ubuntu0.18.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384,
bits: 256, compression: off)
Type "help" for help.


bookr=>
```

6. Execute a basic diagnostic SQL command to show information about the database you are connected to. Type in **`SELECT current_catalog;`** and then press *Enter*.

   You will see output like the following, which shows the current database name is **`bookr`**:

```
bookr=> SELECT current_catalog;
 current_catalog
-----------------
 bookr
(1 row)
```

7. Now that you have confirmed that the database and user were created correctly, you can exit **`psql`** by pressing *Ctrl* + *D*. Typing this will enter the command **`\q`** (for quit) and then exit:

```
bookr=> \q
ben@bookr:~$
```

   You should now be back at the Linux command prompt.

8. We have finished with the virtual machine again, so you can shut it down if you want, or leave it running if you plan to continue with the next exercise soon.

In this exercise, we set up a PostgreSQL user with the **`createuser`** command. Using the **`createdb`** command, we then created a database that is owned by this user. After doing that, we confirmed that the user could authenticate to the database by connecting using the **`psql`** database shell and executing a test command to show the current database.

In the next section, we will discuss how to set up other system users and how to transfer our code onto the virtual machine.

SYSTEM USERS

When we installed Ubuntu, we created a user for the system, and we have been using it to log in and run commands. Linux supports multiple users on the same machine. When deploying Django applications onto a Linux machine, the best practice is to have different system users owning and executing each application. This will segregate the applications' security so that if one user's account is compromised, the attacker will not have access to the other users' accounts. These users will also have limited permissions, so they won't be able to use **`sudo`** or even log in at all.

Initially, it might seem a bit confusing that Ubuntu has two commands that can be used to add users: **useradd** and **adduser**. Now, **useradd** is lower-level, while **adduser** is easier to use and can be run interactively to prompt for options. **adduser** actually calls **useradd** to perform the process of adding the user to the system. In this chapter, we will be using **adduser**.

adduser can be called with just a username and it will prompt for all other required information. The information can also be provided as arguments on the command line for non-interactive user creation. This is how we'll add the user that will run the Bookr application.

The flags and options that we'll use are:

- **--disabled-password**: Creates a user with an invalid password, which makes it impossible to log in using a password.
- **--shell /bin/false**: The default shell (command prompt) on Ubuntu is **bash** (**Bourne Again SHell**). Here, we specify that, instead of bash, the user's shell should be the program called false (inside the **/bin** directory); this is a command that immediately exits with an error status. Even if somehow the user were able to log in, they would immediately be logged out since their "shell" terminates before doing anything.
- **--gecos Bookr**: It would take a bit of a history lesson to explain the naming of this flag (**GECOS** stands for **General Electric Comprehensive Operating System**, a very early Linux ancestor from the 1960s). Essentially, this is some profile information about the user. In our case, we are setting the user's real name to **Bookr**. Without supplying the flag, we'd be prompted to enter this information.

Putting these all together, the command to create a user is:

```
adduser --disabled-password --shell /bin/false --gecos Bookr bookr
```

Where the last argument (**bookr** in lowercase) is the username. Next, we will briefly cover Linux groups and permissions, and how they relate to file ownership.

GROUPS AND PERMISSIONS

Each Linux user may belong to one or more groups. When a user is created (such as with the **`adduser`** command), they are placed into a group with the same name – the user ben would be in a group also called **`ben`**, for example. Groups can be used to determine extra permissions for a user. The main user you created during Ubuntu installation (in *Exercise 17.02*, *Ubuntu Linux Installation*) was also placed into the **`sudo`** group, which allows it to use the **`sudo`** command.

The user and group system also allow for an extensible method of setting the access permissions of files and folders. In Linux, each file and folder has both an owning user and group assigned to it.

Then, it has three sets of accesses assigned: one for the owner of the file, one for users in the group that owns the file, and one for other users. The access that any one of these groups of users may have to the file can be any combination of read (**`r`**), write (**`w`**), and execute (**`x`**). Note that for directories, the execute access mode means being allowed to change into the directory using **`cd`**. A directory may be readable but not executable, which means that you would be able to list files inside it but not enter it.

These permissions are set numerically, with a binary flag for each access type. See *Figure 18.1* for an illustration:

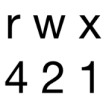

Figure 18.1: Read, write, and execute permissions and their numeric equivalents

The **`x`**, for execute, is the least significant bit and has the value **`1`**. **`w`**, for write, has the value **`2`**. **`r`**, for read, has the value **`4`**.

In this way, an access mode can have a value between **`0`** and **`7`**, inclusive. To illustrate with some examples:

- The value **`0`** is **`000`** in binary and would indicate that the access is false for read, write, and execute.
- The value **`7`** is **`111`** in binary, so read, write, and execute are all true.
- A value of **`4`** (**`100`** in binary) indicates a read-only access mode, while **`5`** (**`101`** in binary) is true for read and execute – this is a common mode for programs on the system.

- These numeric permissions are assigned to files or directories in groups of three, in the order user, group, then other.
- A file with permission **777** can be read, written, and executed by anyone, whereas a file with permission **000** would not be readable, writable, or executable by anyone.
- The permission **755** means the owner can read, write, and execute while the group and user can only read and execute. A permission of **640** would allow the owner to read and write the file, the group to read, and everyone else to not have access.

We need to know these numeric permissions to set them with the **chmod** program – we will cover this soon. When listing directories (using **ls**) with the **-l** flag (to show extra information), we see the permissions of files explicitly listed rather than numerically. We can also see the owning user and group. For example, to look inside the **/home** directory, you can try the following command:

```
$ ls -l /home
total 4
drwxr-xr-x 4 ben   ben   4096 May 12 10:08 ben
```

The directory listing information is output in columns; the permissions are in the first column: **drwxr-xr-x**. The first character, **d**, indicates that this is a directory and is not actually a permission flag. The next three characters represent the owning **user** permissions – this is **rwx** or **111** in binary (**7** in decimal). The three characters after those are the owning **group** permissions, **r-x**. This corresponds to **101** in binary or **5** in decimal, meaning that anyone in the owning group can read the directory or list it but not write to it. The final three characters are also **r-x** so those same permissions apply to all other users.

The next column, **4**, we won't explain in detail as it's not too important. It's the number of items that link to this directory. The column after (**ben**) is the owning user, and the one after, also **ben**, is the owning group. Next is the size of the directory's metadata (**4096** bytes) – this is not the size of the contents of the directory. Next is the last modification time of the directory, and finally, the name of the directory.

We'll look at one more example to illustrate permissions: the **/etc/sudoers** file, which contains information about who can run **sudo**:

```
$ ls -l /etc/sudoers
-r--r----- 1 root root 755 Jan 18  2018 /etc/sudoers
```

The permissions are **-r--r-----**. Since it starts with **-**, this means it is a regular file. Next, the owner permissions are **r--** or **4** (**100** in binary), so the owner can only read the file. Likewise with the group permissions. The other permissions are **---** or **0**, meaning that no one else can even read the file. The owning user and group are both **root** and the file is 755 bytes in size.

We now know how to interpret the current permissions of files, but they can also be set. As we mentioned, the permissions (or mode) of a file can be modified using the **chmod** program (short for CHange MODe). The owning user or group can be set using the **chown** command (short for CHange OWNer). While **chmod** and **chown** can be used in complex ways, we will be using them in a very simple manner in this book.

chmod takes two arguments: the first is the numeric permissions to set on a file, and the second is the filename or relative path. We'll show an example – first, to set the permissions on a file to **777** (full access). Here is the **ls** output of the file before:

```
-rw-rw-r-- 1 ben ben 8 May 12 11:04 example-file.txt
```

Then we execute the command **chmod 777 example-file.txt**.

Here is the file afterward:

```
-rwxrwxrwx 1 ben ben 8 May 12 11:04 example-file.txt
```

> **NOTE**
>
> The precise output will differ based on the username and the name of the file.

Next, we'll remove group and other permissions, and set the owner permissions to just read and write (**6**): **chmod 600 example-file.txt**

This is how the **ls** output looks now:

```
-rw------- 1 ben ben 8 May 12 11:04 example-file.txt
```

Now let's briefly see some examples of **chown**. This command takes a user, group, or both as the first argument and the filename or path as the second argument. If providing a user and group, they should be separated with a colon. When providing just a group, it should be preceded by a colon. A username can be used by itself. We'll see all of this in use in the example that follows.

To change the owning group of a file, the user must both own the file and be a member of the target group. For example, the user **ben** is in the **sudo** group, so with the following command, we can change the owning group to **sudo**:

```
chown :sudo example-file.txt
```

The **ls** output is now:

```
-rw------- 1 ben  sudo    8 May 12 11:04 example-file.txt
```

Changing to other groups will give the message **chown: changing group of 'example-file.txt': Operation not permitted**.

Usually, only the **root** user can change the owner of a file (since a user can't "be" both the owner and the new user), so we execute **chown** commands using **sudo**. To change the owner of the example file to the **bookr** user, we'll have to run:

```
sudo chown bookr example-file.txt
```

The output of **ls** now shows this:

```
-rw------- 1 bookr sudo 8 May 12 11:04 example-file.txt
```

Notice that the group has not updated. It should also be noted that the user that owns a file does not need to belong to the group that owns the file. You can't tell this from the **ls** output, but we do know that **bookr** is not in the **sudo** group.

Finally, we can set the user and group back to what we started with (**ben**/**ben**) using the following command:

```
sudo chown ben:ben example-file.txt
```

And **ls** shows the user and group have both been set back to **ben**:

```
-rw------- 1 ben ben 8 May 12 11:04 example-file.txt
```

chmod and **chown** can both be executed with the **-R** flag to recursively apply the permission or ownership to all items in a directory.

In the next exercise, we will create the **bookr** user on the system, then set up the directory structure and permissions to properly upload and serve media and static files. We will use the **adduser**, **chmod**, and **chown** commands to achieve this, along with the **mkdir** command to make the directories.

EXERCISE 18.02: ADDING SYSTEM USERS AND DIRECTORIES

In this exercise, you will start by creating a system user called **bookr**. Then, you will create the directories to serve and store static and media files. Finally, you will update the ownership and permissions of these directories to ensure that the user **bookr** can write to them and the **www-data** group can read them.

1. Start VirtualBox and then start up your virtual machine. Let it finish booting up – you'll know it's finished because you'll see the login prompt.
2. SSH into your virtual machine using the IP address you found in *Exercise 17.03*, *VirtualBox Networking Configuration* (or **localhost**).
3. Once you have connected, create the **bookr** user using the **adduser** command. Run this command to do that:

```
sudo adduser --disabled-password --shell /bin/false --gecos Bookr
bookr
```

   As we explained in the previous section, this will create a user called **bookr** who is unable to log in.

   You should see output like the following:

```
ben@bookr:~$ sudo adduser --disabled-password --shell /bin/false
--gecos Bookr bookr
[sudo] password for ben:
Adding user `bookr' ...
Adding new group `bookr' (1001) ...
Adding new user `bookr' (1001) with group `bookr' ...
Creating home directory `/home/bookr' ...
Copying files from `/etc/skel' ...
ben@bookr:~$
```

   Note that if you are continuing from the previous exercise, you might not be prompted for your password for **sudo**.

4. Now create the **static** and **media** directories that will serve static files. On Ubuntu, all web server data is stored in the **/var/www** directory. The first step is to create a **bookr** parent directory that will contain both the **static** and **media** directories. Create it with the **mkdir** command. You must use **sudo** to run these commands as the **/var/www** directory is not writable for our non-root user. Run the command:

```
sudo mkdir /var/www/bookr
```

The command doesn't give any output unless there are errors:

```
ben@bookr:~$ sudo mkdir /var/www/bookr
ben@bookr:~$
```

Now create the **`static`** directory in the same manner. Run this command:

```
sudo mkdir /var/www/bookr/static
```

Once again, this command gives no output if successful:

```
ben@bookr:~$ sudo mkdir /var/www/bookr/static
ben@bookr:~$
```

Finally, create the **`media`** directory with this command:

```
ben@bookr:~$ sudo mkdir /var/www/bookr/media
ben@bookr:~$
```

No output indicates a successful creation.

5. Now give the new directories the right ownership. Since we created them using **`sudo`**, they are owned by **`root`**. We'll update the ownership of each directory separately. First change the user and group ownership of **`/var/www/bookr`** to **`bookr`**, using this command:

```
sudo chown bookr:bookr /var/www/bookr
```

Like **`mkdir`**, no output is given if the command is successful:

```
ben@bookr:~$ sudo chown bookr:bookr /var/www/bookr
ben@bookr:~$
```

Next **`/var/www/bookr/media`** and **`/var/www/bookr/static`** both need to be owned by the **`bookr`** user and www-data group. We can update both directories with a single command:

```
sudo chown bookr:www-data /var/www/bookr/media /var/www/bookr/static
```

Again, there should be no output:

```
ben@bookr:~$ sudo chown bookr:www-data /var/www/bookr/media /var/www/
bookr/static
ben@bookr:~$
```

6. The final step to setting up these directories is to ensure the owner (**bookr**) has full access, the group (**www-data**) has only read access, and others have no access. This corresponds to the permission **750**. For the **/var/www/bookr** parent directory, since its group is not **www-data**, we'll give it permission **755** so anyone can read it. Since it only has the **static** and **media** directories directly inside, this is not a big security risk.

   Do these changes using **chmod**. First, the parent directory, by running this command:

```
sudo chmod 755 /var/www/bookr
```

   There is no output:

```
ben@bookr:~$ sudo chmod 755 /var/www/bookr
ben@bookr:~$
```

   Then, the **media** and **static** directories can be updated at the same time, as they have the same access required:

```
sudo chmod 750 /var/www/bookr/media /var/www/bookr/static
```

   There will be no output again:

```
ben@bookr:~$ sudo chmod 750 /var/www/bookr/media /var/www/bookr/
static
ben@bookr:~$
```

7. Validate the directories were created correctly and have the correct ownership using the **ls -al** command, and listing **/var/www/bookr**:

```
ben@bookr:~$ ls -al /var/www/bookr
total 16
drwxr-xr-x 4 bookr bookr    4096 May 13 02:16 .
drwxr-xr-x 4 root  root     4096 May 12 21:47 ..
drwxr-x--- 2 bookr www-data 4096 May 13 02:16 media
drwxr-x--- 2 bookr www-data 4096 May 12 21:53 static
ben@bookr:~$
```

   The entry **.** represents **/var/www/bookr**, and we can see its ownership is **bookr:bookr**, with permissions **drwxr-xr-x**. The **media** and **static** directories are owned by **bookr:www-data** with permissions **drwxr-x--**.

We can also validate the permissions by trying to **ls** the directories we don't have access to; for example, trying to list the media directory as our main user:

```
ben@bookr:~$ ls /var/www/bookr/media/
ls: cannot open directory '/var/www/bookr/media/': Permission denied
ben@bookr:~$
```

Whereas if we try to list the same directory as the **bookr** user, we can see it works fine without any errors:

```
ben@bookr:~$ sudo -u bookr ls /var/www/bookr/media/
ben@bookr:~$
```

The directory is empty so there is no extra output.

8. We've finished with the virtual machine for now, so you can shut it down if you want, or leave it running if you plan to continue with the next exercise soon.

In this exercise, we created the directories to store the static and media files that Django will use. We used the **mkdir** command to create the directories and **chown** and **chmod** to set their permissions. The NGINX web server does not know how to use these directories yet and we will configure it later to read files from them.

Now that we know the directory structure and database credentials, we can add these to **settings.py**. We'll do this in the next section.

SETTINGS FOR PRODUCTION

In *Chapter 15*, *Django Third Party Libraries*, we introduced the **django-configurations** third-party application that allowed us to use classes to define different settings for dev and production. Our **settings.py** file was updated to read some of the variables from the environment, while some were still hardcoded into the file.

When considering how to store secret settings for production, there are many ways to do it. You might use a service like **etcd** (https://etcd.io/) to store values in a way that can be queried by different services. Or, you might store them in your continuous integration system and have it automatically build a configuration file for you. While there are many options to consider, there is one that should be avoided: you should never store secret values and credentials in the **settings.py** file itself.

The main reason for this is to avoid accidentally pushing your potentially private credentials to source control where they could be seen by the wrong people. By having them in a separate file, they can be ignored by your source control system (for example, by being added to the **`.gitignore`** file), which will prevent them from being uploaded.

We also choose to have them in a separate file because we want to store them in a simple key-value format that many different systems can understand. We want to interpret the file using **`bash`** to load the variables into the environment. When we begin serving the application using Gunicorn, **`systemd`** will read this file as well to set its environment variables too. For this reason, we can't store the variables in a Python file.

This file can have any arbitrary name but should be something easy to recognize, such as **`production.conf`**. Its values are strings, set in simple key-value pairs, for example:

```
DJANGO_SECRET_KEY="abc\"123"
```

Note that double quotes are escaped with a backslash (**`\`**).

Recall also in *Chapter 15*, *Django Third Party Libraries*, that the **`manage.py`** file was updated to read the settings from the new class-based settings. When working with Django as a developer, the normal entry point is **`manage.py`**. However, when Django is served by a web server application (such as Gunicorn), it will be interfaced using **`wsgi.py`**. As such, the **`wsgi.py`** file must also be updated to read the new style settings. This is similar to the process of updating **`manage.py`**: replace the default Django functions with those provided by **`django-configurations`**.

First, a default configuration class should be defined. Since we're intending the **`wsgi.py`** file to only be used in production, it can default to **`Prod`**:

```
os.environ.setdefault('DJANGO_CONFIGURATION', 'Prod')
```

Consider the line with the default Django import:

```
from django.core.wsgi import get_wsgi_application
```

The preceding line is replaced, with this:

```
from configurations.wsgi import get_wsgi_application
```

In this way, we're using **`get_wsgi_application`** from **`django-configurations`** instead of the one provided by Django.

We updated the **DATABASES** and **SECRET_KEY** settings to be read from environment variables in *Chapter 15*, *Django Third Party Libraries*. In the next exercise, we will also update **settings.py** to read **MEDIA_ROOT** and **STATIC_ROOT** from the environment, as these need to be set to those that we created in *Exercise 18.02*, *Adding System Users and Directories*. We'll also make the changes to **wsgi.py** as just described.

THE REQUIREMENTS FILE

Before moving onto the next exercise, we need to cover the **requirements.txt** file. If you're not familiar with it, it's a method of recording all the packages that are currently installed in your Python virtual environment. You can use **pip** to *freeze* the list of packages and write them to a text file. Later, you can copy the text file to another computer and reinstall all the same packages in a new virtual environment.

To generate this list of packages, simply run **pip3 freeze** (or simply **pip freeze** on Windows) in the virtual environment for which you want the list of installed packages. For example, in the current Bookr virtual environment:

```
$ pip3 freeze
dj-database-url==0.5.0
Django==3.0.3
django-configurations==2.2
…
```

Or, if you're using Windows, the command is just **pip freeze**.

Note that some of the package output has been left out for brevity. There are about 14 packages in total.

To write this to a file, use the output redirect **>** character:

```
$ pip3 freeze > requirements.txt
$
```

Remember, in Windows, this will be **pip freeze**:

```
$ pip freeze > requirements.txt
$
```

There is no output, but if you examine **requirements.txt**, you should see it contains the same list of packages as above.

To install requirements from the requirements file, run **`pip3 install`** with the **`-r requirements.txt`** argument. If we try this in the existing virtual environment, we'll see that nothing is installed since, of course, the requirements already match since we just generated the file:

```
$ pip3 install -r requirements.txt
…
Requirement already satisfied: Django==3.0.3 in /Users/ben/.virtualenvs/
bookr/lib/python3.7/site-packages (from -r requirements.txt (line 3))
(3.0.3)
…
$
```

We'll see what happens when we try to install packages in an empty environment in the next exercise.

Packages can manually be added to a requirements file too. For example, when we set up Bookr on our virtual machine, we will need to use the **`psycopg2`** package to connect to PostgreSQL. This can sometimes be difficult to install on non-Linux operating systems, so instead of installing it and including it using **`pip3 freeze`**, we will just type it into **`requirements.txt`** after it has been generated.

UPLOADING FILES TO THE VIRTUAL MACHINE

As well as providing a text shell/command prompt, SSH also allows files to be transferred. We can use an **SFTP** (**Secure File Transfer Protocol**) client to connect to our virtual machine and upload files. We'll need to upload the Bookr source code and some configuration files. There are several SFTP clients available, and you might already have one you use (please note that SFTP is different from **FTP**, although some clients support both protocols). If you are already familiar with SFTP, feel free to skip this step and continue with the rest of the exercise. Otherwise, we'll be using FileZilla, which is a free SFTP client. It can be downloaded from https://filezilla-project.org/download.php?type=client. Once it is installed on your computer, carry on to the next exercise and we'll give instructions on how to use it to connect and upload files.

In the next exercise, we will update the **`settings.py`** file to read some more settings from environment variables and create a configuration file containing these variables. We also need to update the **`wsgi.py`** file to read the new-style configuration that **`django-configurations`** provides. We'll then *freeze* the requirements into a text file using **`pip3`**, and finally, upload all these and the Bookr source code to our virtual machine.

EXERCISE 18.03: CONFIGURATION, REQUIREMENTS, AND UPLOADS

In this exercise, you will start by updating your **settings.py** file to read **MEDIA_ROOT** and **STATIC_ROOT** from environment variables. Then, you will create a **production.conf** file containing these settings, along with the database URL, **ALLOWED_HOSTS**, and **SECRET_KEY**. You'll then **freeze** your requirements to the **requirements.txt** file. You need to manually add the **psycopg2** and **gunicorn** requirements to this file as well. The last step is to upload this all to our virtual machine and check that it has been copied properly:

1. On your local computer, launch PyCharm, then open **settings.py** in the **bookr** directory.

2. Scroll to near the bottom of the file and locate the line that sets **STATICFILES_DIRS**. It will be inside the **Dev** class. Above it, add a setting to read **STATIC_ROOT** from an environment variable, like this:

```
STATIC_ROOT = values.Value()
```

   This will make **STATIC_ROOT** be read from the **DJANGO_STATIC_ROOT** environment variable, defaulting back to empty if not set.

3. Now change the **MEDIA_ROOT** setting. Currently, it's a static string, set like this:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

   Change it to be set like this:

```
MEDIA_ROOT = values.Value(os.path.join(BASE_DIR, 'media'))
```

   MEDIA_ROOT will be read from the **DJANGO_MEDIA_ROOT** environment variable if it is set. Otherwise, it falls back to the existing value.

   We're done with **settings.py** now so you can save and close it.

4. In the root **bookr** project directory, create a new file named **production.conf**. It will open automatically. You'll enter the settings here, one per line. The first line you'll add is one to tell **django-configurations** which configuration class to use – simply the value **Prod**:

```
DJANGO_CONFIGURATION="Prod"
```

5. Next, you need to set a secret key. Any 50-character random string should be fine. You can generate a random one using Django by running this in the terminal:

```
python3 -c "import django.utils.crypto; print(django.utils.
crypto.get_random_string(50, 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijkl
mnopqrstuvwxyz0123456789\!@#\$%^&*()-='))"
```

   Put the value you generated into the **`production.conf`** file like this:

```
DJANGO_SECRET_KEY="EZaI^PksI6ISi7BkNxG=2yYiav*=#FN&g)7Of4gy@-
VcSQkdU5"
```

6. Next, put in the database URL. Enter this on the next line:

```
DJANGO_DATABASE_URL="postgres://bookr:password@localhost/bookr"
```

   However, make sure you replace **`password`** with the password you used for the **`bookr`** PostgreSQL user.

7. Next, add these lines to set the media and static directories:

```
DJANGO_STATIC_ROOT="/var/www/bookr/static"
DJANGO_MEDIA_ROOT="/var/www/bookr/media"
```

8. The final setting to add is **`ALLOWED_HOSTS`**, which is a list of the hostnames that are allowed to be used to access your site. This will be the IP address or hostname you have been using to connect over SSH, something like:

```
DJANGO_ALLOWED_HOSTS="192.168.0.123"
```

   Or:

```
DJANGO_ALLOWED_HOSTS="localhost"
```

   Note that even though **`localhost`** and **`127.0.0.1`** usually mean the same thing, make sure you're consistent with what you use. Or, to be safe, you could add both:

```
DJANGO_ALLOWED_HOSTS="localhost,127.0.0.1"
```

   When deploying to a hosted virtual server, you will use the domain name(s) that you have chosen for your site.

   You should now have six lines in this file. You can save and close it.

Open **wsgi.py** inside the **bookr** package directory. You'll need to add the default environment setting of **DJANGO_CONFIGURATION** to **Prod**. This will go after the existing **os.environ.default** call to set **DJANGO_SETTINGS_MODULE**:

```
os.environ.setdefault('DJANGO_CONFIGURATION', 'Prod')
```

9. Next, you must replace and move the **get_wsgi_application** import line. Remove this line:

```
from django.core.wsgi import get_wsgi_application
```

And instead insert this line, after the **os.environ.setdefault** calls:

```
from configurations.wsgi import get_wsgi_application
```

After completing *step 8*, and this one, **wsgi.py** should contain this code (as well as some comments at the start):

```
import os

os.environ.setdefault('DJANGO_SETTINGS_MODULE'),\
                      ('bookr.settings')
os.environ.setdefault('DJANGO_CONFIGURATION', 'Prod')

from configurations.wsgi import get_wsgi_application

application = get_wsgi_application()
```

You can now save and close **wsgi.py**.

10. Now generate the requirements file. In a terminal, change into the **bookr** project directory and make sure the **bookr** virtual environment is active. Then, freeze the installed packages by running this command:

```
pip3 freeze > requirements.txt
```

Or, if you're running Windows:

```
pip freeze > requirements.txt
```

This will write the list of packages out to the **requirements.txt** file.

11. Switch back to PyCharm and open **`requirements.txt`**:

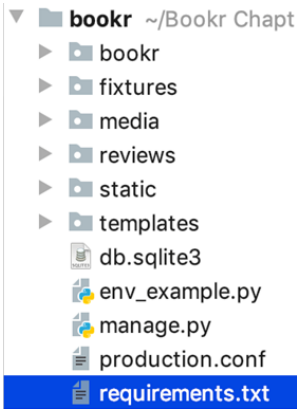

Figure 18.2: requirements.txt in the bookr project directory

You should see it contains a list of all the packages and their versions. For example, **`Django==3.0.3`**, **`django-configurations==2.2`**, and more. They are listed alphabetically but the order doesn't matter. As mentioned earlier, **`psycopg2`** can be hard to install so you will need to add it here manually. Also, since you're already editing this file, add **`gunicorn`** as a requirement so it doesn't need to be manually installed when you deploy the Bookr application. Add these two lines to the end of **`requirements.txt`**:

```
gunicorn==20.0.4
psycopg2==2.8.5
```

Your **`requirements.txt`** file should look similar to *Figure 18.3*; however, some of the versions may differ from the time of writing:

```
1  asgiref==3.2.3
2  dj-database-url==0.5.0
3  Django==3.0.3
4  django-configurations==2.2
5  django-crispy-forms==1.8.1
6  django-debug-toolbar==2.2
7  django-rest-framework==0.1.0
8  djangorestframework==3.11.0
9  Pillow==7.0.0
10 pytz==2019.3
11 six==1.14.0
12 sqlparse==0.3.0
13 gunicorn==20.0.4
14 psycopg2==2.8.5
```

Figure 18.3: requirements.txt contents

You can now save and close **`requirements.txt`**, and you will be finished with PyCharm for now.

12. Next, upload the Bookr project directory to the virtual machine. First, make sure **`VirtualBox`** has been started and your virtual machine has finished booting up. Then, start FileZilla, and you'll be shown the **`FileZilla`** window (*Figure 18.4*):

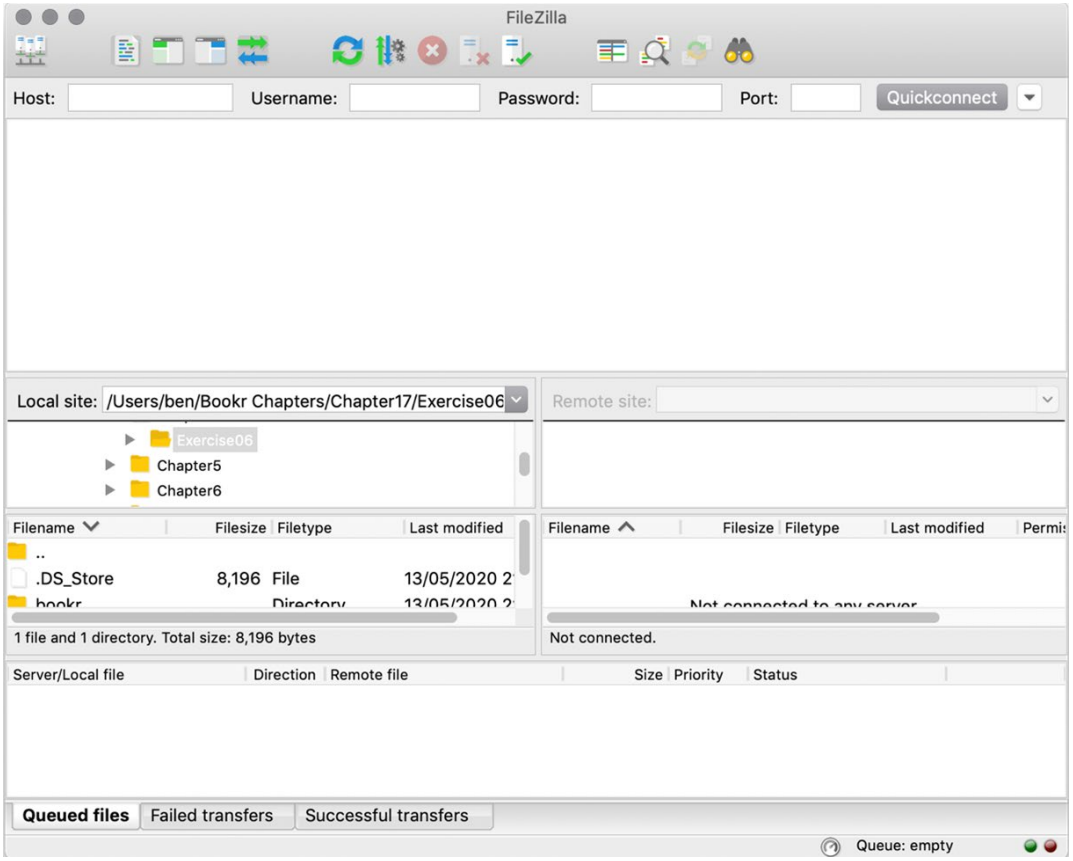

Figure 18.4: FileZilla window

The bar near the top with **`Host`**, **`Username`**, **`Password`**, and **`Port`** fields is the **`Quickconnect`** bar.

13. To connect to your virtual machine, **`Host`** should be the IP address you have been using for SSH, prefixed by **`sftp://`**; for example, **`sftp://192.168.0.123`** or **`sftp://127.0.0.1`**. **`Username`** and **`Password`** are also those you use to connect over SSH. After entering all those, click **`Quickconnect`** to connect.

14. On the first connection, you'll be prompted to verify the server's host key (*Figure 18.5*):

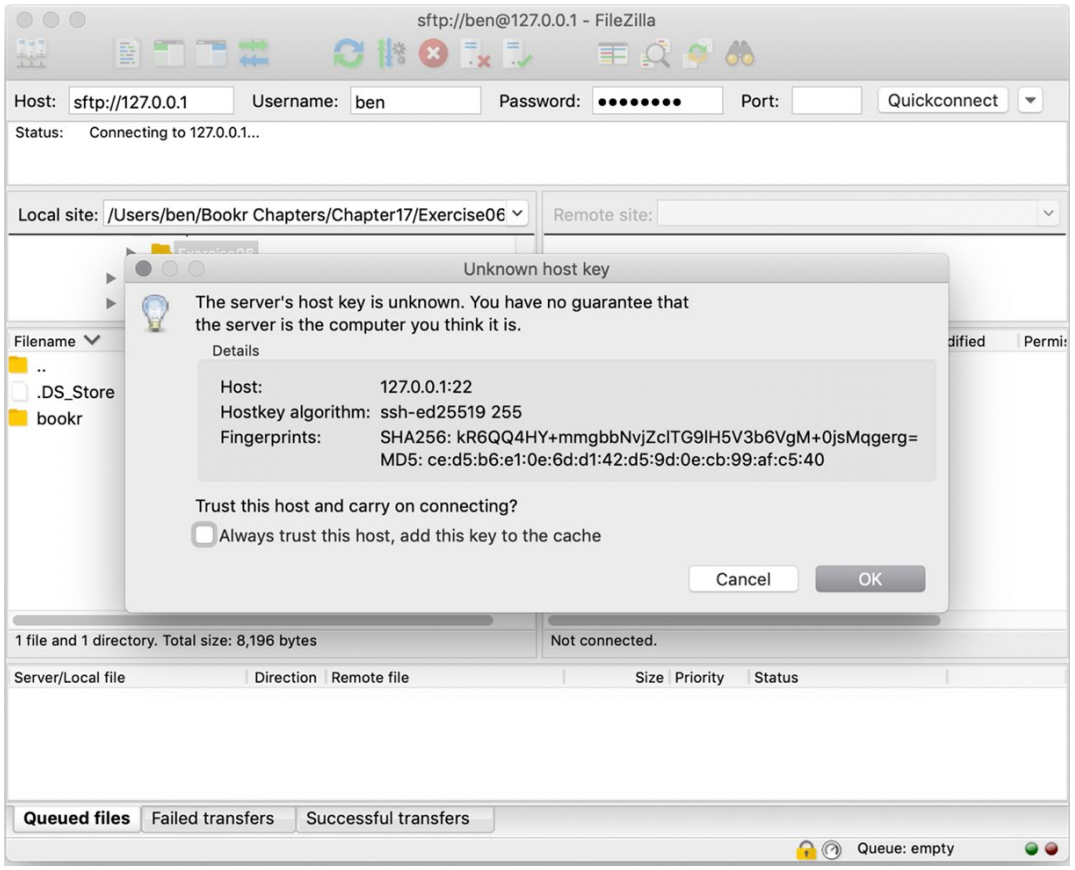

Figure 18.5: Host key verification

Check the **`Always trust this host, add this key to the cache`** checkbox so you won't be prompted next time, then click **`OK`**.

15. After connecting, your files are shown in the left pane, and the server's files are in the right pane. You can upload files or folders by dragging from the left pane to the right. Navigate to the **`bookr`** directory on your computer in the left pane, then drag it to the right side inside your home directory, that is, **`/home/<yourusername>`**.

After the upload is complete, you should see the **`bookr`** directory on the right, indicating that it uploaded successfully (*Figure 18.6*):

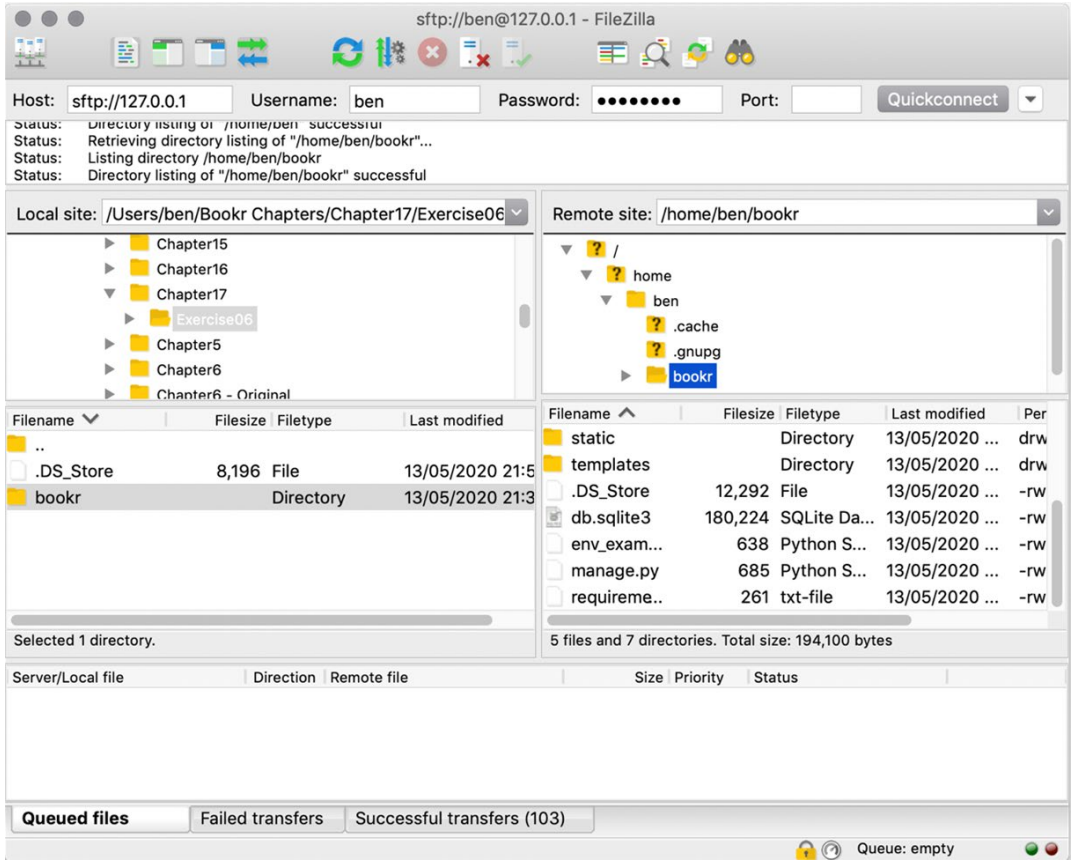

Figure 18.6: Uploaded files shown in FileZilla

We're finished with FileZilla for now, so you can quit it.

16. You can also verify that the upload was successful by connecting to the virtual machine using SSH. Then, run the **`ls`** command to list the contents of your home directory:

```
ben@bookr:~$ ls
bookr
ben@bookr:~$
```

Then, list the **bookr** directory and you should see all the Bookr files:

```
ben@bookr:~$ ls bookr
bookr       env_example.py  manage.py  production.conf   reviews
templates
db.sqlite3  fixtures        media      requirements.txt  static
ben@bookr:~$
```

17. You're finished with SSH and the virtual machine for now, so shut down the virtual machine (or, as usual, leave it running if you will be continuing with the next exercise soon).

In the next section, you will copy the **bookr** directory into the **bookr** user's home directory. Then you'll set up a virtual environment and install the Python requirements.

MOVING FILES AND UPDATING PERMISSIONS

In the previous exercise, we copied the Bookr data onto the virtual machine; however, it was copied into the home directory of the user that was created when setting up the virtual machine, not the **bookr** user. The reason is that the **bookr** user is not allowed to log in and so it could not upload the files. Your normal user can log in and upload files but can't write into the **bookr** home directory. The solution is, therefore, to upload as your normal user then move the files into the **bookr** user's directory after logging in using SSH. Then, the files need to be updated to be owned by the **bookr** user. This is done with the **chown** command. Finally, for extra security, you should also update the permissions on the **production.conf** file so other users can't read it – permission mode **600**.

We can demonstrate this process by showing the commands that must be run. All these commands must be run using **sudo**. After logging into the virtual machine running SSH, first, move the **bookr** directory into the **bookr** user's home directory:

```
ben@bookr:~$ sudo mv bookr /home/bookr/
[sudo] password for ben:
ben@bookr:~$
```

Note that **sudo** is prompting for the password since it's the first time it's been used for a while. Otherwise, there would be no output.

Then, update the ownership of the **bookr** directory and all the files/directories it contains – we can use the **-R** flag to do this recursively:

```
ben@bookr:~$ sudo chown -R bookr:bookr /home/bookr/bookr
ben@bookr:~$
```

Then, update the access permission on the production configuration file, so that only **bookr** can read it:

```
ben@bookr:~$ sudo chmod 600 /home/bookr/bookr/production.conf
ben@bookr:~$
```

Now the virtual environment can be set up by the **bookr** user and the requirements can be installed in it.

SETTING UP THE VIRTUAL ENVIRONMENT AND PIP INSTALL

We have already emphasized the fact that **bookr** cannot log in using SSH because the user has no password. However, we can still run commands as **bookr** using **sudo**. Since the next commands must be run as **sudo**, it can be tedious to have to prefix **sudo** every time. That's why we will switch to the **bookr** user using **sudo**.

One way to do that is to execute the **bash** shell as **bookr**, for example:

```
ben@bookr:~$ sudo -H -u bookr /bin/bash
[sudo] password for ben:
bookr@bookr:/home/ben$ pwd
/home/ben
bookr@bookr:/home/ben$ cd
bookr@bookr:~$ pwd
/home/bookr
bookr@bookr:~$
```

We use the **-H** flag so the home directory of the user is reset to that of the **sudo** target (without it, the shell would still think the home directory for **bookr** was **/home/ben**). The prompt updates to show that we are now **bookr@bookr** and not in our **bookr** home directory. Typing **cd** with no argument takes us back to **bookr** home directory. We can confirm this by then running **pwd** to show we are indeed in **/home/bookr**.

Assuming we have now switched to the correct user (**bookr**), we can execute the **virtualenv** Python module to create a new virtual environment. Since we already have a directory called **bookr**, we'll pick a different name – **booker-venv** is fine:

```
bookr@bookr:~$ python3 -m virtualenv --python=python3 bookr-venv
Already using interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /home/bookr/bookr-venv/bin/python3
Also creating executable in /home/bookr/bookr-venv/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
bookr@bookr:~
```

> **NOTE**
>
> Note that since we're not going to be interacting with the virtual environment much, we don't have to use any helper tools such as **virtualenvwrapper** that can simplify virtual environment management.

Next, we can activate the virtual environment using the **source** command:

```
bookr@bookr:~$ source bookr-venv/bin/activate
(bookr-venv) bookr@bookr:~$
```

Notice that the prompt updates to indicate the virtual environment is active.

Finally, the requirements can be installed from the **requirements.txt** file using the **-r** flag:

```
(bookr-venv) bookr@bookr:~$ pip3 install -r bookr/requirements.txt
Collecting asgiref==3.2.3
  Downloading asgiref-3.2.3-py2.py3-none-any.whl (18 kB)
Collecting dj-database-url==0.5.0
  Downloading dj_database_url-0.5.0-py2.py3-none-any.whl (5.5 kB)
Collecting Django==3.0.3
  Downloading Django-3.0.3-py3-none-any.whl (7.5 MB)
…
Successfully built django-rest-framework psycopg2
Installing collected packages: asgiref, dj-database-url, …
(bookr-venv) bookr@bookr:~$
```

Now all the requirements are installed, we can test this by trying to run **`django-admin.py`**:

```
(bookr-venv) bookr@bookr:~$ django-admin.py

Type 'django-admin.py help <subcommand>' for help on a specific
subcommand.
# rest of the output is truncated
(bookr-venv) bookr@bookr:~$
```

Now that Django is set up, database migration and static collection can be performed.

RUNNING MIGRATIONS AND COLLECTSTATIC

Performing Django migrations and running **`collectstatic`** should now be familiar to you, having seen them earlier in this book. However, we need to do one more thing before running them. Earlier, we suggested placing all your environment variables in a configuration file so they could be loaded into your environment before running commands. Before you can use **`manage.py`**, the **`production.conf`** settings must be exported to your current session.

First, change into the **`bookr`** directory (note that **`bookr-venv`** is still active). Then, load the config with this command:

```
export $(cat production.conf | xargs)
```

It will read the **`production.conf`** file and export (set inside our shell) each variable it finds:

```
(bookr-venv) bookr@bookr:~$ cd bookr
(bookr-venv) bookr@bookr:~/bookr$ export $(cat production.conf | xargs)
(bookr-venv) bookr@bookr:~/bookr$
```

Then the management commands can be run as we normally would. First, **`migrate`**:

```
(bookr-venv) bookr@bookr:~/bookr$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, reviews, 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
  -
(bookr-venv) bookr@bookr:~/bookr$
```

Then **`collectstatic`**:

```
(bookr-venv) bookr@bookr:~/bookr$ python manage.py collectstatic

176 static files copied to '/var/www/bookr/static'.
(bookr-venv) bookr@bookr:~/bookr$
```

If we now list **`/var/www/bookr/static`**, we should see the static file namespace directories are there:

```
(bookr-venv) bookr@bookr:~/bookr$ ls /var/www/bookr/static
admin  debug_toolbar  rest_framework  reviews
(bookr-venv) bookr@bookr:~/bookr$
```

To switch back to our normal user, we can type *Ctrl* + *D* or use the **`exit`** command.

In the next section, we'll put the final preparations of the Python setup into practice by moving the **`bookr`** source code, changing its ownership, then setting up a virtual environment and installing requirements. Then, we'll be able to do the standard Django project setup steps, such as running the migrations and collecting the static files.

EXERCISE 18.04: PYTHON VIRTUAL ENVIRONMENT AND DJANGO SETUP

In this exercise, you will move the Bookr files into the **`bookr`** user's directory, and then update their ownership so they are owned by the **`bookr`** user and group. Then, you'll create a virtual environment and install the requirements into it. Finally, you'll be able to execute Django to apply the database migrations and collect the static files:

1. Start VirtualBox and then start up your virtual machine if it's not already running. Let it finish booting up – you'll know it's finished because you'll see the login prompt.
2. SSH into your virtual machine using the IP address you found in *Chapter 17*, *Deployment of a Django Application (Part 1 – Server Setup)* (or **`localhost`**).
3. Move the **`bookr`** project directory into the **`bookr`** home directory, using the **`mv`** command. Since your user won't have permission to do this, it must be run with **`sudo`**:

```
    sudo mv bookr /home/bookr/
```

There is no output, except to prompt for your password if necessary:

```
ben@bookr:~$ sudo mv bookr /home/bookr/
[sudo] password for ben:
ben@bookr:~$
```

4. Next, update the ownership of all **bookr** code to be owned by the **bookr** user and group. Use the **chown** command with the **-R** (recursive) flag:

```
sudo chown -R bookr:bookr /home/bookr/bookr
```

Again, no output indicates success:

```
ben@bookr:~$ sudo chown -R bookr:bookr /home/bookr/bookr
ben@bookr:~$
```

5. The last change before switching to the **bookr** user is to change the access mode on **production.conf** to **600** using **chmod**:

```
sudo chmod 600 /home/bookr/bookr/production.conf
```

There should be no output:

```
ben@bookr:~$ sudo chmod 600 /home/bookr/bookr/production.conf
ben@bookr:~$
```

This will prevent anyone but **bookr** from reading or writing to it.

6. Now switch to the **bookr** user with this command:

```
sudo -H -u bookr /bin/bash
```

Then, change to the **bookr** home directory with the command cd (no arguments):

```
ben@bookr:~$ sudo -H -u bookr /bin/bash
bookr@bookr:/home/ben$ cd
bookr@bookr:~$
```

7. Create a **bookr** virtual environment by executing the **virtualenv** Python module:

```
python3 -m virtualenv --python=python3 bookr-venv
```

This will create the **bookr-venv** directory:

```
bookr@bookr:~$ python3 -m virtualenv --python=python3 bookr-venv
Already using interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /home/bookr/bookr-venv/bin/python3
Also creating executable in /home/bookr/bookr-venv/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
bookr@bookr:~$
```

8. Now activate the virtual environment using the **source** command with the **activate** script in the virtual environment directory:

```
source bookr-venv/bin/activate
```

Then, the requirements can be installed with **pip3** by using the **-r** flag and providing the path to the requirements file (**bookr/requirements.txt**).

The requirements will be downloaded and installed; you will see output like the following:

```
bookr@bookr:~$ source bookr-venv/bin/activate
(bookr-venv) bookr@bookr:~$ pip3 install -r bookr/requirements.txt
Collecting asgiref==3.2.3
  Downloading asgiref-3.2.3-py2.py3-none-any.whl (18 kB)
Collecting dj-database-url==0.5.0
  Downloading dj_database_url-0.5.0-py2.py3-none-any.whl (5.5 kB)
…
Installing collected packages: asgiref, dj-database-url, …
(bookr-venv) bookr@bookr:~$
```

9. You can now use Django, but first, you need to export the settings to the environment. **cd** into the **bookr** directory, then run this:

```
export $(cat production.conf | xargs)
```

Here's an example output:

```
(bookr-venv) bookr@bookr:~$ cd bookr
(bookr-venv) bookr@bookr:~/bookr$ export $(cat production.conf |
xargs) (bookr-venv) bookr@bookr:~/bookr$
```

There will be no output, but you can test it worked by echoing (printing out) one of the variables, for example:

```
(bookr-venv) bookr@bookr:~/bookr$ echo $DJANGO_STATIC_ROOT
/var/www/bookr/static
(bookr-venv) bookr@bookr:~/bookr
```

10. Now you can apply the Django database migrations. This is done in the usual way, with the **`manage.py migrate`** command:

```
(bookr-venv) bookr@bookr:~/bookr$ python3 manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, reviews, 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
  …
(bookr-venv) bookr@bookr:~/bookr$
```

11. The final Django setup is to collect the static files, with the **`collectstatic`** management command:

```
python3 manage.py collectstatic
```

You might be prompted to confirm that you want to want to overwrite existing files, as in this next example. Type **`yes`** and then press *Enter*:

```
(bookr-venv) bookr@bookr:~/bookr$ python3 manage.py collectstatic

You have requested to collect static files at the destination
location as specified in your settings:

    /var/www/bookr/static

This will overwrite existing files!
Are you sure you want to do this?

Type 'yes' to continue, or 'no' to cancel: yes

179 static files copied to '/var/www/bookr/static'.
(bookr-venv) bookr@bookr:~/bookr$
```

You can confirm this worked by listing **`/var/www/bookr/static`**:

```
(bookr-venv) bookr@bookr:~/bookr$ ls /var/www/bookr/static
admin          logo.png  recent-reviews.js  reviews
debug_toolbar  main.css  rest_framework
(bookr-venv) bookr@bookr:~/bookr$
```

12. We're finished with SSH and the virtual machine for now, so disconnect and shut down the virtual machine (or, as usual, leave it running if you will be continuing with the next exercise soon).

In this exercise, we moved the **`bookr`** project files to be under the **`bookr`** user and updated their ownership. We then set up a **`bookr`** virtual environment and installed all the requirements. Once that was done, we had a working Django installation, so we were able to apply the database migrations and collect the static files.

We're almost at the end of our deployment. In the next section, we will configure **Gunicorn** to host the application and NGINX to serve static files and proxy connections.

GUNICORN CONFIGURATION OPTIONS

`gunicorn` has many options that can be used to configure how it should run, and it can be tuned to your server configuration. These can be set in a configuration file or as command-line flags. We will use just configuration files for altering settings in this book. A common configuration option to set is how many worker processes should be started. If your server has more CPU cores, then it can handle more workers without becoming overloaded. In general, this should be two to four workers per CPU core.

In this chapter, we will give sensible defaults for configuration values. The Gunicorn documentation goes into detail on how to choose your settings based on the type of application you're serving and your server configuration. The *Design* document (https://docs.gunicorn.org/en/latest/design.html#design) gives some theoretical information on the types of workers and the *Settings* documentation (https://docs.gunicorn.org/en/latest/settings.html) expands on the available settings.

GUNICORN/SYSTEMD CONFIGURATION

In *Chapter 17*, *Deployment of a Django Application (Part 1 – Server Setup)*, we introduced **Gunicorn** and showed how easy it was to serve a Django application using a single command:

```
gunicorn gunicorn bookr.wsgi:application
```

Since we have Gunicorn installed in the virtual environment, we could run this command now and have a running WSGI server. The problem is that it would quit as soon as we disconnected our SSH session.

To address this problem, we need to set up Gunicorn to run as a service. On Ubuntu, services are managed by software called **`systemd`**. You have been using this without realizing it during the exercises in this chapter: one of the services that **`systemd`** has started on the virtual machine is the SSH server. Users can write their own configuration files for **`systemd`** to configure how services are started. For example, these files can specify the users that execute the server, which environment variables should be set, how crashes should be handled (that is, should the service restart automatically), and in what order the service starts (for inter-service dependency).

`systemd` has three types of configuration files: services, sockets, and targets. To briefly summarize:

- A **`service`** file is a running process or processes as we described in the previous paragraph.
- A **`socket`** file is a network or file socket that **`systemd`** will listen on, and then only start a service when the socket has activity.
- A **`target`** file is a group of related services. This can be used to set up dependencies so services in one target won't start until the target on which it depends is ready.

To set up Gunicorn, we'll only need to write a **`service`** file and a **`socket`** file. To describe the configuration, we'll show the files we'll be using for Bookr and then explain the options. First, here's the **`systemd`** service file. It is saved at **`/etc/systemd/system/gunicorn-bookr.service`** and should look like this:

```
[Unit]
Description=Book Gunicorn daemon
Requires=gunicorn-bookr.socket
After=network.target

[Service]
```

```
EnvironmentFile=/home/bookr/bookr/production.conf
Type=notify
User=bookr
Group=bookr
RuntimeDirectory=gunicorn-bookr
WorkingDirectory=/home/bookr/bookr
ExecStart=/home/bookr/bookr-venv/bin/gunicorn bookr.wsgi:application
ExecReload=/bin/kill -s HUP $MAINPID
KillMode=mixed
TimeoutStopSec=5
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

The lines in square brackets separate the configuration into stanzas. **`[Unit]`** is sort of like metadata about the service:

- **`Description`** is used to describe the service for diagnostics.
- **`Requires`** tells **`systemd`** what our service requires. In this case, it must have our socket set up before it can start.
- **`After`** provides information as to when the service should start. In this case, we should start after the network is up.

The **`[Service]`** section describes how the service is started:

- **`EnvironmentFile`** is a file that contains configuration data (environment variables) that should be loaded when starting the service.
- **`Type`** specifies how the process is started and how it communicates its status to **`systemd`**.
- **`User`** and **`Group`** specify which user and group the process will run as. We use the **`bookr`** user and group.
- **`RuntimeDirectory`** is the name of the directory in which **`systemd`** will store information about the running process, such as its process ID.
- **`WorkingDirectory`** is the directory the process will change to when starting – we make it the **`bookr`** project directory.
- **`ExecStart`** sets the actual command that's run to start the service. We provide the full path to **`gunicorn`** in our virtual environment and the module path of the WSGI application.

- **`ExecReload`** tells **`systemd`** what command to run to reload the service. Without going into too much detail, in this case, it will **`kill`** the process using its ID, and ask it to restart.
- **`KillMode`** and **`TimeoutStopSec`** specify how the service is stopped and then how long **`systemd`** waits until forcibly stopping it.
- **`PrivateTmp`** is a Boolean. If set to **`true`**, then the service will have its own **`/tmp`** (temporary directory) not shared with other services.

`[Install]` specifies properties that apply when the service is installed. Its one setting, **`WantedBy`**, specifies that our service is "wanted by" **`multi-user.target`**. This means when **`systemd`** loads the **`multi-user`** target, it should include our service.

> **NOTE**
>
> The multi-user target basically means the normally booted Linux system.

Note that throughout the configuration file, we have used **`bookr`** where possible, for example, **`Requires=gunicorn-bookr.socket`** and **`RuntimeDirectory=gunicorn-bookr`**. By doing this, we can host multiple Django (or any WSGI applications) on the same server by creating new configuration files and replacing **`bookr`** with the name of the new application.

Next, we'll look at the socket file. It is much shorter. It is saved at **`/etc/systemd/system/gunicorn-bookr.socket`** and looks like this:

```
[Unit]
Description=Bookr Gunicorn Socket

[Socket]
ListenStream=/run/gunicorn-bookr.sock
User=www-data
Mode=600

[Install]
WantedBy=sockets.target
```

`Description` serves the same purpose as in **`service`**.

The **[Socket]** stanza describes the socket configuration:

- **ListenStream** is the path to the socket. This will be the socket through which NGINX and Gunicorn communicate. We will set the same path in the NGINX configuration.
- **User** is the user who will own the socket. It is set to **www-data** since NGINX is also run by the **www-data** user and will be writing to the socket.
- **Mode** is the access mode of the socket. **600** means only **www-data** will be able to read and write data; all other users have no access.

The final **[Install]** stanza is also like the service file. The socket is **WantedBy** the **sockets** target (group).

After the service and socket files are in place, **systemd** can be told to enable the configuration with the following command:

```
sudo systemctl enable --now gunicorn-bookr.socket
```

Note that we only specify that the socket should be enabled. Since we have the line **Requires=gunicorn-bookr.socket** in the service file, the service will be enabled too. Running this command will both start the service and set up **systemd** to start the service at boot time.

After Gunicorn is started, we still need a web server that proxies requests to it. In the next section, we will discuss how to configure **NGINX**, which is the last thing to do before we see our site running on our virtual machine.

NGINX CONFIGURATION

NGINX is already configured to run as a service. This was set up as part of the install process using apt. We saw it was running by loading its example web page in *Exercise 17.04*, *Package Update and Installation*. This means we don't have to set up **systemd** configuration for it. We do, however, have to write a configuration file for NGINX itself.

NGINX configuration files are stored in the **/etc/nginx** directory. Some files contain global NGINX settings, such as **/etc/nginx/nginx.conf**. NGINX also splits configurations for specific sites into their own files, and they are stored in **/etc/nginx/sites-available**. It comes with one **default** site set up, which is configured at **/etc/nginx/sites-available/default**.

NGINX doesn't actually read from the **/etc/nginx/sites-available** directory. It reads from **/etc/nginx/sites-enabled**, which is a directory containing symlinks to files inside the **sites-available** directory.

A **symlink**, short for symbolic link, is a file that points to another path on disk. If the target (the file or directory that is pointed to) is deleted, then the symlink will still exist on disk. However, if you try to read/write to the symlink path, you'll run into an error. On the contrary, the symlink can be removed without affecting the target file.

For example, the file **`/etc/nginx/sites-enabled/default`** is a symlink to **`/etc/nginx/sites-available/default`**. This is done so that sites can be turned off and on by deleting and recreating the symlinks, without destroying the original configuration file.

Now we'll have a look at the NGINX config file that we'll be using. Since it's quite a long file, we'll look at it in sections. The configuration options are grouped inside curly braces. The name of the setting is the first word on a line, and the setting value comes after a space. The setting is terminated with a semicolon. For example, this line sets the **`server_name`** value to **`localhost`**:

```
server_name localhost;
```

Onto the configuration file. The first part of the configuration defines an "upstream" server. An upstream is a named list of servers that can be used to handle requests. In our case, it's called **`django`**. The upstream can be configured with a single server or a group of servers that will be cycled through to handle requests. We'll only be using one upstream server:

```
upstream django {
    server unix:/run/gunicorn-bookr.sock fail_timeout=0;
}
```

Inside the **`upstream`** configuration group, our **`server`** is the socket that's used to communicate with Gunicorn. By specifying **`fail_timeout=0`**, we'll continue to use Gunicorn even if it generated an error. Otherwise, after one exception in our Python code, we'd have to restart NGINX.

The next configuration block is called **server**. You can think of each server block as a configuration for a virtual website that NGINX is hosting. We introduced the concept of virtual hosting in *Chapter 1*, *Introduction to Django*. Web servers use the HTTP **`Host`** header in the HTTP request to marshal requests. We would generally have one server block per hostname that we were hosting (although a server can be configured to accept multiple names – so that we could serve the same content for both **`example.com`** and **`www.example.com`**).

The first **server** block is quite simple but illustrates the concept:

```
server {
    listen 80 default_server;
    return 444;
}
```

We want the server to **listen** on port **80**. The **default_server** argument means this server should handle any hostname not handled by other server definitions. All this server does is return an **HTTP 444**, which is a special code that indicates that NGINX should just close the connection without sending any data. This is to prevent malicious clients from tying up resources on our server by making requests for invalid hostnames.

Note that while this might look like a function with a **return** at the end, it's just a directive. The order of the configuration doesn't matter.

Next, we'll look at the **server** block that will serve Bookr. This will be examined in chunks:

```
server {
    listen 80 deferred;
```

It starts in a similar way as the previous **server** block, listening on port **80**. The **deferred** option allows better performance by changing how the connection is accepted:

```
    server_name localhost;
```

This is the hostname the server will respond to. It should match the hostname you have been using to connect to the server, as well as whatever is in Django's **ALLOWED_HOSTS** setting. Multiple hostnames can be set with spaces; for example:

```
server_name localhost 127.0.0.1;
```

When you're deploying your site to a hosted virtual server, this will be the domain name(s) that you have chosen to host your site on:

```
    root /var/www/bookr;
```

root is the directory on the disk that NGINX considers to be the root directory for the server. We've set it to **/var/www/bookr**, so requests for a path such as **/static/logo.png** will map to the file **/var/www/bookr/static/logo.png**:

```
    client_max_body_size 4G;
    keepalive_timeout 5;
```

These two settings configure more about the HTTP connection. The default upload size for NGINX is **1MB**. The first setting, **client_max_body_size**, sets this to **4GB** instead. We also set **keepalive_timeout** to 5 seconds, so if no data is transferred for 5 seconds, a keepalive connection will be closed:

```
    location / {
      try_files $uri @proxy_to_django;
    }
```

Inside a **server** block, we can specify different rules for different locations. Having this stanza begin with **location /** means the settings will apply to the entire site – unless they are overridden by more specific **location** blocks.

The **try_files** option tells NGINX that for any given request, it should try to load the file from disk using its path. If it is not found, it should fall back to the **@proxy_to_django** location, which we'll look at soon.

To use the same example as before, a request to **http://<hostname>/static/logo.png** will be located on disk at **/var/www/bookr/static/logo.png** and NGINX will serve the file directly. Compare this to a request for **http://<hostname>/books/**. NGINX will check for the **/var/www/bookr/books** file on disk. Since it doesn't exist, the request will be passed on to the **@proxy_to_django** location. Here is the configuration for the **@proxy_to_django** virtual location:

```
    location @proxy_to_django {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_pass http://django;
    }
}
```

Since we're proxying the request, we need to add some headers that Django otherwise wouldn't receive, using the **proxy_set_header** setting. The **X-Forwarded-For** header contains the original IP address of the remote client. We need to add this otherwise Django would only see connections from a local socket. **X-Forwarded-Proto** is how Django knows whether the site was accessed using HTTP or HTTPS. Finally, **Host** is forwarded on unchanged – we already know Django checks this against the **ALLOWED_HOSTS** setting.

The **`proxy_redirect`** setting defines how to redirect to the proxy server. Since we're not performing a redirect to the upstream server – instead we're just passing the request internally – we set **`proxy_redirect`** to **`off`**. The final line, **`proxy_pass`**, sets where the request is passed to. The value **`http://django`** refers to the **`upstream`** server that was defined at the start of the file.

To summarize, the following parts of the configuration are most important (which also provides an overview of how NGINX handles the request):

- The definition of an **`upstream`** server – this contains the information about Gunicorn and which socket to connect to it with.
- The **`server`** block, which defines an address to listen to and accept the HTTP connection.
- The **`server root`** setting, which defines the path on disk that static files are read from.
- The use of the **`try_files`** option: NGINX will first try to load the file from disk and if it doesn't exist, then it will fall back to the specified location and pass the request on to there.
- The virtual location (**`@proxy_to_django`**) specifies that an upstream server should be used – the same one we specified at the start of the configuration.

After updating the NGINX configuration, NGINX needs to be reloaded to read in the new configuration. This is done using the **`systemctl`** command:

```
sudo systemctl reload nginx
```

`systemctl` is a system control tool that can start, stop, and restart many services on your virtual machine. For example, the following command will fully restart NGINX instead of just reloading the configuration:

```
sudo systemctl restart nginx
```

This command will stop a service – **`gunicorn-bookr`** in this case:

```
sudo systemctl stop gunicorn-bookr
```

And this command starts a service again (**`gunicorn-bookr`**):

```
sudo systemctl start gunicorn-bookr
```

Note that **`gunicorn`** should be restarted after making changes to your Python code. You should use the name of the service you created when restarting (that is, **`gunicorn-bookr`**). No services need to be restarted if you're just changing static files or templates.

That was a brief introduction to NGINX configuration. It (and the Gunicorn **`systemd`** configuration) was based on the *Deploying Gunicorn* guide at https://docs.gunicorn.org/en/stable/deploy.html. NGINX has many more configuration options to suit different systems, but they are beyond the scope of this book. Now it is time for the final exercise and to finally see Bookr running on your virtual machine.

In the next exercise, you will create **`systemd`** and NGINX configuration files, then upload them to your virtual machine. At the end of the exercise, you'll be able to see Bookr running under a production-like system.

EXERCISE 18.05: GUNICORN AND NGINX CONFIGURATION

In this exercise, you will write the configuration files for Gunicorn (**`systemd`**) and NGINX, then upload them to the virtual machine. You will then move them to the **`/etc`** directory and change their ownership. Then, you will disable the default NGINX configuration by deleting the symlink. Finally, you will enable the Gunicorn service and reload NGINX, then see Bookr running on your virtual machine:

1. You'll need to store configuration for these services in the project directory. In PyCharm, create a new directory named **`service-conf`** inside the Bookr project directory:

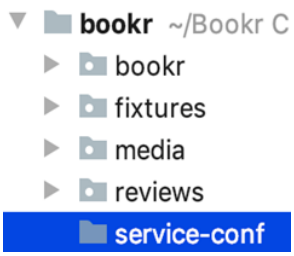

Figure 18.7: The service-conf directory inside the bookr project directory

2. Create a file named **`gunicorn-bookr.service`** inside this new directory. PyCharm will ask you how you want to treat **`*.service`** files by showing a window called **`Register New File Type Association`**:

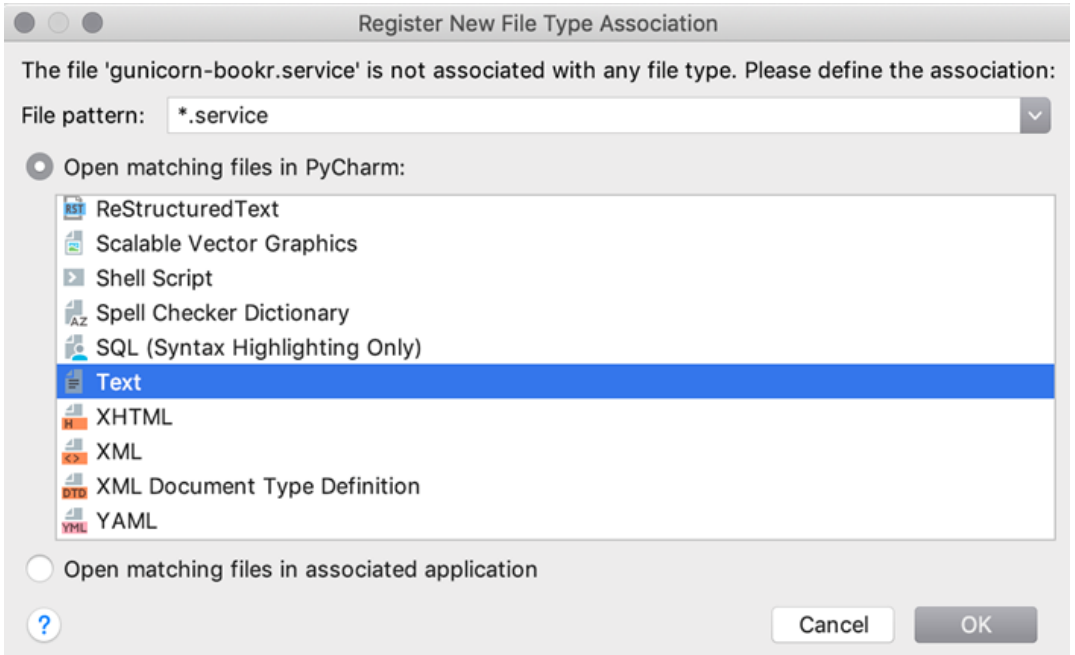

Figure 18.8: Register New File Type Association window

Select **`Text`** and click **`OK`**.

3. The new file will open. Enter this content:

```
[Unit]
Description=Book Gunicorn daemon
Requires=gunicorn-bookr.socket
After=network.target

[Service]
EnvironmentFile=/home/bookr/bookr/production.conf
Type=notify
User=bookr
Group=bookr
RuntimeDirectory=gunicorn-bookr
WorkingDirectory=/home/bookr/bookr
ExecStart=/home/bookr/bookr-venv/bin/gunicorn bookr.wsgi:application
ExecReload=/bin/kill -s HUP $MAINPID
```

```
KillMode=mixed
TimeoutStopSec=5
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

You learned what this configuration means in the **Gunicorn/Systemd Configuration** section. You can save and close the file.

4. Create another file in the **service-conf** directory named **gunicorn-bookr.socket**. You will be asked how to treat ***.socket** files, and once again you should select **Text**, then click **OK**.
5. The new file will open. Enter this content:

```
Description=Bookr Gunicorn Socket

[Socket]
ListenStream=/run/gunicorn-bookr.sock
User=www-data
Mode=600

[Install]
WantedBy=sockets.target
```

Similar to **gunicorn-bookr.service**, you learned about this file in the *Gunicorn/Systemd Configuration* section. You can now save and close the file.

6. Create one last file in the **service-conf** directory, called just **bookr**. This will be the NGINX config file. You'll need to tell PyCharm to treat it as text as well. Enter this config data into the new file:

```
    upstream django {
        server unix:/run/gunicorn-bookr.sock fail_timeout=0;
    }

    server {
        listen 80 default_server;
        return 444;
    }


    server {
```

```
            listen 80 deferred;

            server_name localhost;

            root /var/www/bookr;

            client_max_body_size 4G;
            keepalive_timeout 5;

            location / {
              try_files $uri @proxy_to_django;
            }

            location @proxy_to_django {
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_
    for;
                proxy_set_header X-Forwarded-Proto $scheme;
                proxy_set_header Host $http_host;
                proxy_redirect off;
                proxy_pass http://django;
            }
        }
```

You must be familiar with this configuration already. You saw it in the section titled *NGINX Configuration*. You can save and close the file.

7. You'll need to copy the config files to the virtual machine. Start VirtualBox and boot up your virtual machine. Once it has finished starting, log into it using FileZilla.

Upload the **`service-conf`** directory to your home directory on the virtual machine:

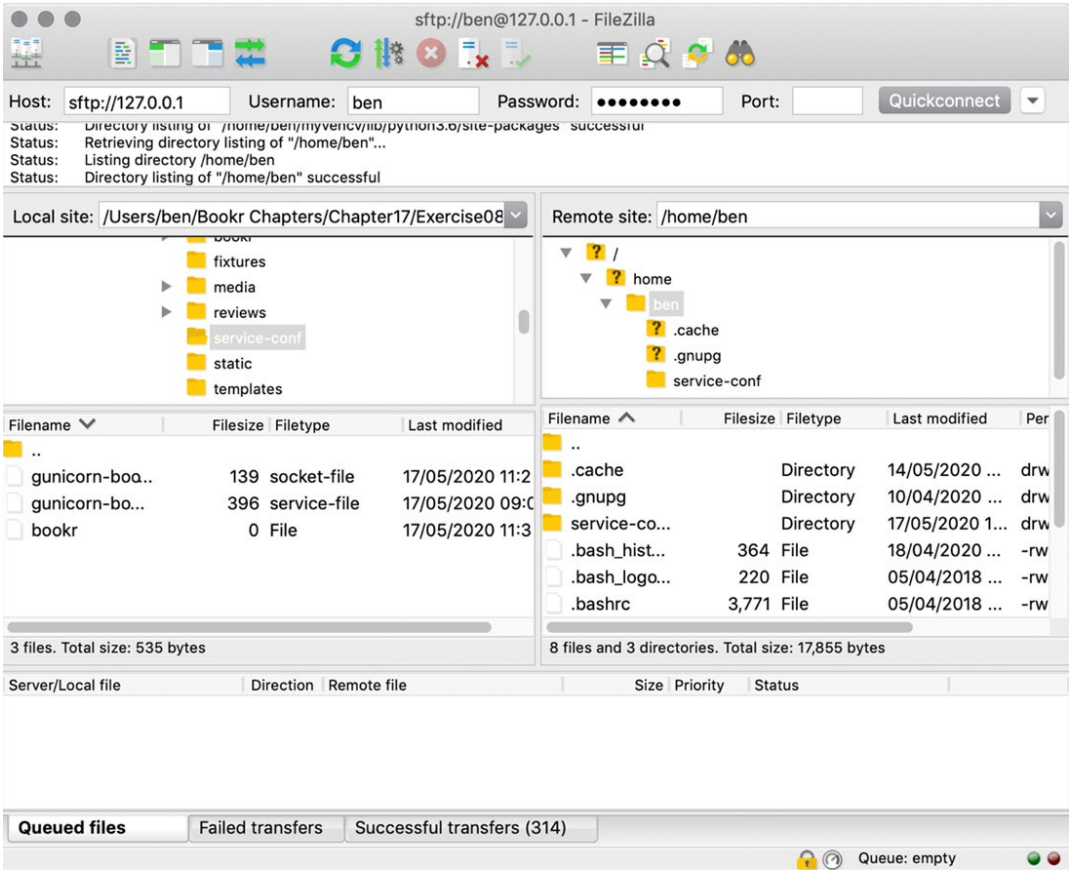

Figure 18.9: The service-conf directory copied to the virtual machine

8. Now move the config files into the right place and update their ownership. SSH into your virtual machine.

   First, move the **`gunicorn-bookr.service`** file to the **`/etc/systemd/system/`** directory, using **`sudo`**:

```
sudo mv service-conf/gunicorn-bookr.service /etc/systemd/system/
```

You may have to enter your password for **sudo**:

```
ben@bookr:~$ sudo mv service-conf/gunicorn-bookr.service /etc/
systemd/system/
[sudo] password for ben:
ben@bookr:~$
```

Likewise, move the **gunicorn-bookr.service** file to the same directory:

```
sudo mv service-conf/gunicorn-bookr.socket /etc/systemd/system/
```

There should be no output:

```
ben@bookr:~$ sudo mv service-conf/gunicorn-bookr.socket /etc/systemd/
system/
ben@bookr:~$
```

Finally, move **service-conf/bookr** to **/etc/nginx/sites-available/**:

```
sudo mv service-conf/bookr /etc/nginx/sites-available/
```

Once again, no output is given:

```
ben@bookr:~$ sudo mv service-conf/bookr /etc/nginx/sites-available/
ben@bookr:~$
```

9. Now that the files are in the right place, you must fix their ownership and permissions. They should all be owned by, and have the group, **root**. This can be applied to all the files with one command:

```
sudo chown root:root /etc/systemd/system/gunicorn-bookr.service /etc/
systemd/system/gunicorn-bookr.socket
```

There will be no output if the command is successful:

```
ben@bookr:~$ sudo chown root:root /etc/systemd/system/gunicorn-bookr.
service /etc/systemd/system/gunicorn-bookr.socket /etc/nginx/sites-
available/bookr
ben@bookr:~$
```

The config files need to have the permission **644**, and we can set this on all the files at once with a single command too:

```
ben@bookr:~$ sudo chmod 644 /etc/systemd/system/gunicorn-bookr.
service /etc/systemd/system/gunicorn-bookr.socket /etc/nginx/sites-
available/bookr
ben@bookr:~$
```

10. You now need to delete the **`default`** NGINX configuration and link your new one into the **`sites-enabled`** directory. First, the removal of **`default`**, with this command:

```
sudo rm /etc/nginx/sites-enabled/default
```

    There will be no output if the command was executed successfully:

```
ben@bookr:~$ sudo rm /etc/nginx/sites-enabled/default
ben@bookr:~$
```

    Remember that this just removes the symlink – the original file is left intact.

    Now you'll create a new symlink with the **`ln`** (link) command using the **`-s`** flag to indicate it should be symbolic. The order of the paths is the target (destination) file first, then the path to the new link to create the second:

```
sudo ln -s /etc/nginx/sites-available/bookr
```

    You will see no output when the command executes successfully:

```
ben@bookr:~$ sudo ln -s /etc/nginx/sites-available/bookr /etc/nginx/
sites-enabled/bookr
ben@bookr:~$
```

11. Finally, enable the Gunicorn services:

```
sudo systemctl enable --now gunicorn-bookr.socket
```

    This produces some output about creating a symlink:

```
ben@bookr:~$ sudo systemctl enable --now gunicorn-bookr.socket
Created symlink /etc/systemd/system/sockets.target.wants/gunicorn-
bookr.socket B /etc/systemd/system/gunicorn-bookr.socket.
ben@bookr:~$
```

    Then reload the NGINX configuration:

```
sudo systemctl reload nginx
```

    This produces no output on success:

```
ben@bookr:~$ sudo systemctl reload nginx
ben@bookr:~$
```

    And you're ready to test that Bookr is up and running on your server.

Open a web browser and navigate to the IP address you've been using throughout this chapter; for example, **`http://locahost/`**, **`http://127.0.0.1/`**, or **`http://192.168.0.123/`**. Note that since we're now running with a standard port number, you don't have to specify any port. If everything was successful, you should see Bookr running:

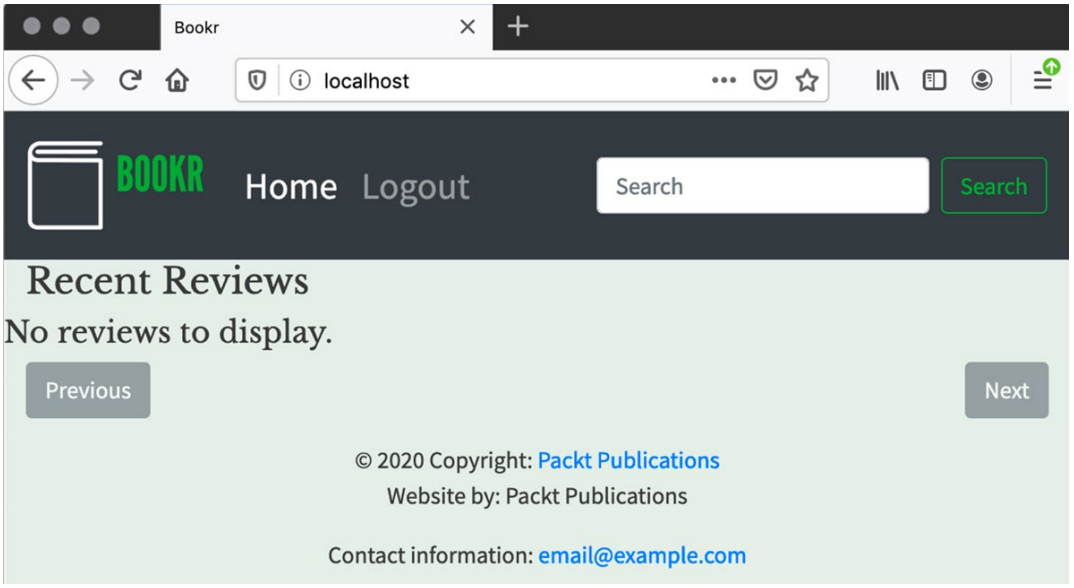

Figure 18.10: Bookr running on the virtual machine

If it didn't work, you may have to check the logs on the server to find out why. The **`less`** command lets you view the logs one page at a time. The log that will contain this information is the Syslog at **`/var/log/syslog`**, or the NGINX error log at **`/var/log/nginx/error.log`**.

For example, use the **`less /var/log/syslog`** command to view the Syslog. After the log is open, you can navigate around it using the *Up*/*Down* arrow keys and *Page Up* and *Page Down*. Type *q* to quit.

12. We've finished with the virtual machine again, so you can shut it down if you don't intend to continue the activities soon. When you reboot it again, NGINX and Gunicorn should start automatically.

In this exercise, you created configuration files for **`systemd`** to start **`gunicorn`**, and configuration files for **`nginx`**. You then uploaded them to your virtual machine and gave them the right ownership and permissions to match existing config files. You removed the default NGINX configuration and sym-linked your new config file to activate it. Finally, you enabled the **`systemd`** configuration and reloaded NGINX. The Bookr site was visible in the browser.

With this, we have completed the final exercise in this book. This chapter was a deep dive into getting Django deployed to a virtual machine, from the very beginning.

You might have noticed that there is no data on the Bookr site that we've deployed. In the next activity, you will put your knowledge into practice by running Django management commands to create a superuser and import the test data.

ACTIVITY 18.01: MANAGEMENT COMMANDS

In this activity, you will run management commands to add some test data as a superuser. You have done these before but only on your local machine, so you will need to connect to the virtual machine, switch users, and activate the virtual environment. When the activity is completed, the test data will be imported and you will have a superuser to connect to the Django admin interface with.

These steps will help you complete this activity:

1. Make sure your virtual machine is running and connect to it with SSH.
2. Switch to the **`bookr`** user.
3. Change to the **`bookr`** user's home directory.
4. Activate the **`bookr`** virtual environment.
5. Load the Bookr project's **`production.conf`** environment variables.
6. Run the **`loadcsv`** management command to import **`WebDevWithDjangoData.csv`**.
7. Create a superuser with the **`createsuperuser`** management command.

After completing these steps, you should be able to view the Bookr website and see the test data (*Figure 18.11*):

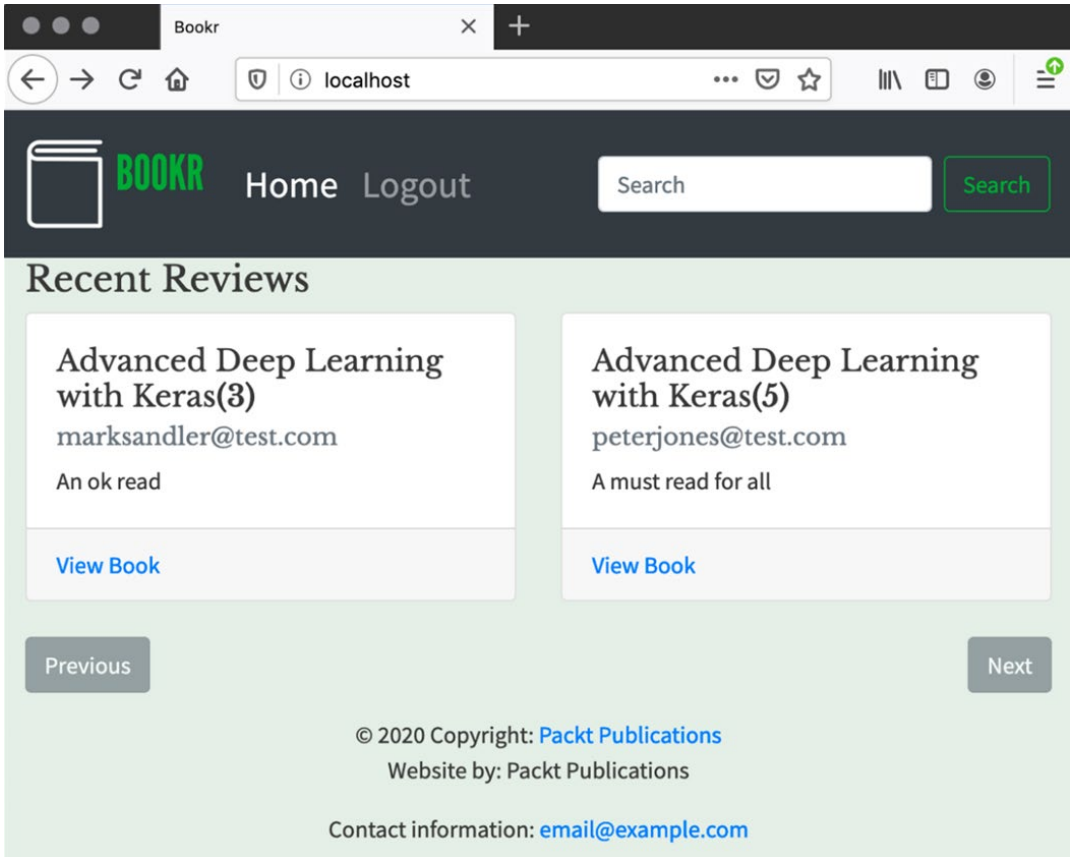

Figure 18.11: Bookr with test data loaded

Then, you should be able to log in to the Django Admin interface to test the user you created is working (*Figure 18.12*).

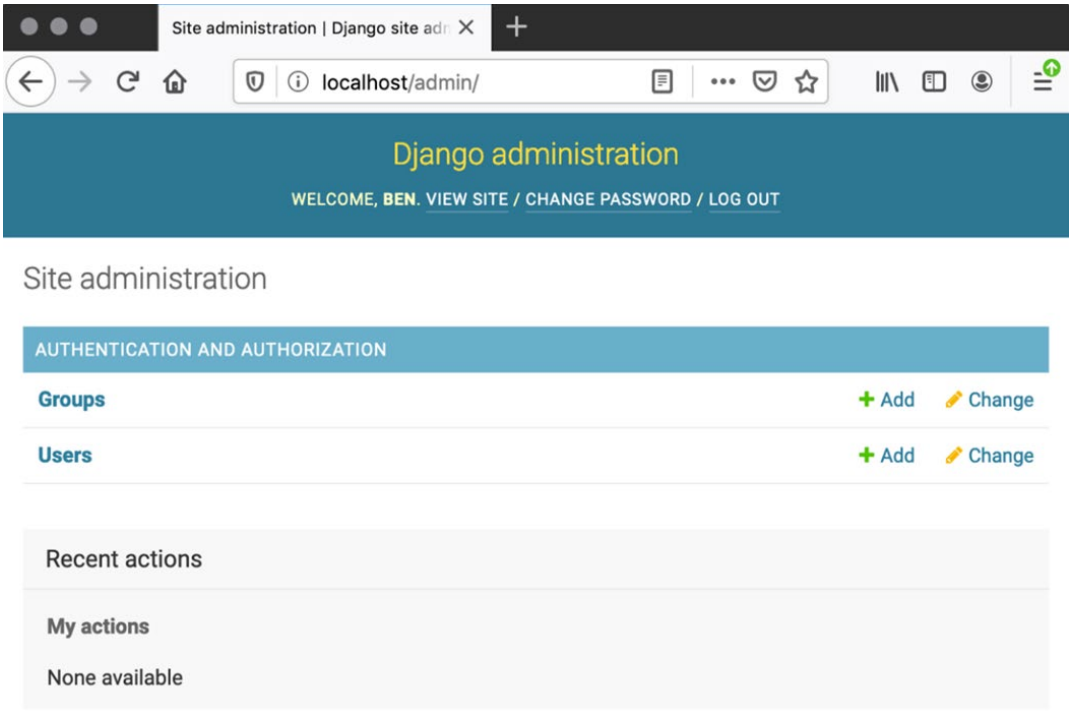

Figure 18.12: Django Admin site after logging in

> **NOTE**
>
> The solution to this activity can be found at http://packt.live/2Nh1NTJ.

ACTIVITY 18.02: CODE UPDATES

In this last activity, you will go through the process of making a code change then deploying it to the server, similar to how you might do it for real site changes.

You have been asked to make a couple of simple changes to Bookr:

- The page background color is to be updated.
- We want to have a welcome message on the main page that simply says, **`Welcome to Bookr!`**.

You can make and test these changes on your computer with the Django dev server, but you will have to then deploy them to the virtual machine too.

These steps will help you complete this activity:

1. Change the **`body background-color`** to **`#f0f0f0`**, in **`main.css`**.
2. In the **`reviews/templates/reviews/index.html`** template, add an **`<h2>`** inside the **`content`** block. It will contain the text **`Welcome to Bookr!`**.
3. After testing your changes with the Django dev server, upload the files to your virtual machine using FileZilla.
4. Move the files to the **`bookr`** project directory under the **`bookr`** user. Update their ownership.
5. Activate the virtual environment and run the **`collectstatic`** management command.
6. Switch back to your normal user. Restart the **`gunicorn-bookr`** service using the **`systemctl`** command.

After completing these steps, you should view Bookr on your virtual machine using your browser. Note that you might have to do a force refresh to make sure the new CSS is downloaded. You should see the welcome message you added, and the new background color:

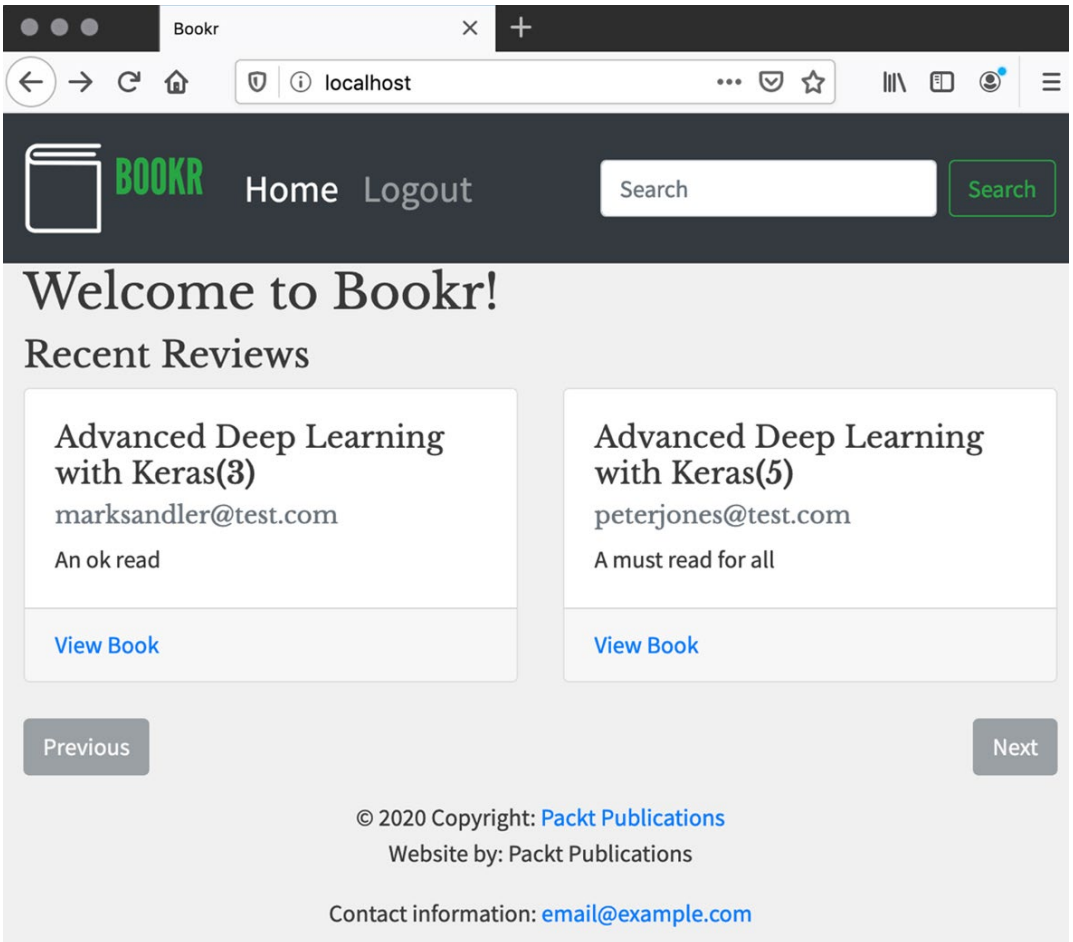

Figure 18.13: Welcome message

> **NOTE**
>
> The solution to this activity can be found at http://packt.live/2Nh1NTJ.

SUMMARY

In this chapter, we finished up the deployment of our Django application, Bookr. We learned about PostgreSQL and why we use it (or other database servers) instead of SQLite. We then set up a PostgreSQL user and database. We learned about Linux system users and permissions, including how to set file ownership and modes with **`chown`** and **`chmod`**, respectively. We used **`pip freeze`** to store a list of Python requirements and then deployed them to our virtual server, in a new virtual environment. We configured our Django application, Gunicorn, NGINX, and **`systemd`** with production settings files, which were uploaded to our virtual machine with SFTP. We used the **`export`** command to set up the environment variables and then used the Django **`manage.py`** commands to configure the database and static files. Finally, we were able to restart NGINX and Gunicorn to get our Django application up and running on our virtual machine in a method similar to production. You should now be able to apply your skills to deploy a Django application to a virtual server hosted by a cloud provider.

The chapter also brings to a close *Web Development with Django*. Congratulations on the amazing journey you've been on. You started out with simple Django views and database interactions and moved on to serving static files, media, REST APIs, and even interactive JavaScript frontends.

If you're looking for a way forward, there's nothing better than working on a Django project from start to finish or joining a team and working with other developers on an existing application. Now, you have the skills you need to do that. *Web Development with Django* has provided a broad base and foundation of skills, and when you start working on more projects, you'll be able to choose which of these you want to focus on and extend further.