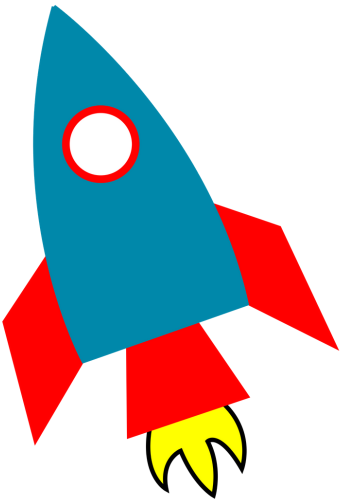
The Unix administration guide for Django developers



Deploying Django

on a single Debian or Ubuntu server

Antonis Christofides

Deploying Django on a single Debian or Ubuntu server
Edition 2.1 (2018-08-13)

The book (and the source code) can be reached through https://djangodeployment.com.

I am grateful to Aisha Bello for a review of the Static and Media Files chapter; to Curtis Maloney for a review of the Gunicorn chapter; and to Markus Holtermann and Chris Pantazis for useful comments.

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Introduction

I want you to understand how Django deployment works, and in order for you to understand it we'll need to experiment. So you will need an experimental Debian or Ubuntu server. You could create a virtual machine on your personal system, but it will be easier and more instructive if you have a virtual machine on the network. So go to Hetzner, Digital Ocean, or whatever is your favourite provider, and get a virtual server. In the rest of this book I will be using $SERVER_IPv4_ADDRESS to denote the ip address of the server on which your Django project is running; so you must mentally replace $SERVER_IPv4_ADDRESS with "1.2.3.4" or whatever the address of your server is. Likewise with $SERVER_IPv6_ADDRESS, if your server has one.

If you find the above confusing, maybe it's because you don't know what this book is about. "Deployment" means installing your Django application on production. This book doesn't teach you how to develop with Django; you need to already know that. If you don't, you need to read another book.

If you are really looking to deploy your Django application, and you can already create a Debian or Ubuntu server, login to it with `ssh`, use `scp` to copy files, use basic commands like `ls`, and understand some basic encryption principles, that is, what is a public and private key, you can probably skip

most of this chapter. Otherwise, I'll take you step by step, right from getting a virtual server, logging in to it, and using essential GNU/Linux commands.

1.2 Getting a server

Until recently, I used to create test servers on my laptop using virtualbox and/or vagrant. However, virtual servers on the cloud have become so cheap that it is usually better to hire one there. It's faster to set up, and you don't need to worry about NAT. The other time I needed a Ubuntu server for a brief test. I created one on DigitalOcean within a couple of minutes; I made my test; and then I destroyed the server, after about half an hour. DigitalOcean's charge for that was $0.01. The cool thing about DigitalOcean is that you can get a test server for only as long as you need it, and get charged only for the number of hours for which the server exists. In other providers you usually pay for the whole month.

(Note: I am not affiliated with DigitalOcean, and I am not using their referrals program.)

So, if you don't already have a cloud VM provider, sign up on https://digitalocean.com and create a droplet. DigitalOcean calls its servers droplets, but they are just virtual machines. In order to create a droplet, you need to choose the operating system and some other things.

If you don't want to know much about your options for the operating system, just choose Ubuntu 16.04 64 bit.

> **Tip:** Debian or Ubuntu?
>
> These two operating systems are practically the same system. You have probably already chosen one of the two to work with, and there is no reason to reconsider.
>
> If you haven't chosen yet, and you want to know nothing about this, go

ahead and pick up the latest LTS version of Ubuntu, which currently is 16.04 (and will continue to be so until April 2018).

The reason I recommend Ubuntu is mostly that it is more popular and therefore has better support by virtual server providers. Ubuntu's Long Term Support versions also have five years of support instead of only three for Debian (though recently Debian has started to offer LTS support but it's kind of unofficial). On the other hand I feel that Ubuntu sometimes rushes a bit too much to get the latest software versions in the operating system release, whereas Debian can be more stable; but this is just a feeling, I have no hard data. I use Debian, but this is a personal preference because sometimes I'm too much of a perfectionist (with deadlines) and I want things my own way.

In Ubuntu's version numbering, the first number is the year and the second is the month; so 16.04 was released in April 2016. The LTS versions are the ones released in April of even years, so the next LTS version will be 18.04. I don't see why someone would use the 32-bit version, which can support only up to 4 GB of RAM, so choose the 64-bit version. **Don't choose a non-LTS version**; support for these lasts less than a year, and it is too little.

Besides operating system, you also need to choose size, data center, IPv6, SSH keys, and host name.

The **size** of the server depends on how heavy the application is. For our purpose, which is testing Django deployment, the smallest one is usually more than enough. In fact, 512 MB of RAM and 20 GB of disk space are sometimes enough for small applications in production.

Choose the **data center** that is nearest to you.

I like my servers to have **IPv6**, so I always turn that on.

Don't specify **SSH keys** yet, unless you are comfortable with them already. I devote the whole next section to SSH keys.

Finally, choose a **host name**. Usually, when it is for testing, I look at the time

and if it's 17:02 I name the server `test1702`. For production, if I don't have anything better, I choose names of Greek rivers at random.

Hit the big green Create button and your server will be ready after one or two minutes. DigitalOcean will create a password for your server and email it to you.

In order to login from **Unix** (such as Linux or Mac OS X), open a terminal and type this:

```
ssh root@[server ip address]
```

The first time you attempt this, it will warn you that the authenticity of the host cannot be established; tell it "yes", you are sure you want to continue connecting. It will then ask for the password. The first time you connect, it may force you to change the password. Note that when you type a password, nothing at all is shown, no bullets or other placeholders, it's as if you are typing nothing, but it is actually registering your keystrokes.

You can logout of the server by entering `exit` at its command prompt. Ctrl+D also works.

From **Windows** you first need to install an SSH client. The most popular one is PuTTY, which you can download from http://putty.org/. It's a single file, `putty.exe`. Each time you execute it, it will launch its configuration window. Type the server ip address in the "Host Name (or IP address)" field and click Open.

The first time you attempt this, it will warn you that the authenticity of the host cannot be established; tell it Yes, you trust the host. It will then ask for the user name ("login as:"), which is `root`, and the password. The first time you connect, the server may force you to change the password. Note that when you type a password, nothing at all is shown, no bullets or other placeholders, it's as if you are typing nothing, but it is actually registering your keystrokes.

Eventually you will want to copy and paste text from and to PuTTY. Just

selecting text automatically copies it to the clipboard, and pasting is just right-clicking.

You can logout of the server by entering `exit` at its command prompt. Ctrl+D also works.

1.3 Introduction to SSH keys

You have deadlines. Learning about SSH keys doesn't seem to be urgent. You can live without them, can't you? Is it worth to spend an hour to learn about them? The answer is yes. If you log on to a server 12 times per day (a conservative estimate), and it takes on the average 5 seconds to type your password (and retype it if it's wrong), that's one minute. You will have paid off your investment in three months. But there are more savings; when creating a droplet on DigitalOcean you will just be ticking a box and you will be ready to login. Otherwise you will be needing to wait for the email to come, copy and paste your password, and go through the process of changing the password. SSH keys can also be used on GitHub and other services. Finally, a little understanding of public key cryptography will later help you setup HTTPS, which is based on the same principles. So let's start.

You will first create a pair of keys, which we call the public key and the private key. Let's just do it first. You won't be understanding what we are doing, but I will explain it afterwards.

On **Unix**, such as Ubuntu or Mac OS X, just enter the command `ssh-keygen`, which stands for ssh key generator. It will ask you a couple of questions:

1. It will ask where to store the keys. Since we are just testing, I suggest to store them in /tmp/id_rsa.
2. It will ask for a passphrase. For the time being, do not use a passphrase. We will come to the passphrase later on.

This will create two files; the private key will be in `/tmp/id_rsa`, and the public key in `/tmp/id_rsa.pub`.

On **Windows**, download `PuTTYgen` from the PuTTY download page. Like `PuTTY`, `PuTTYgen` is a single `.exe` file which you double-click on and it runs. Click on "Generate". It will ask you to move the mouse over the blank area; do so. After it finishes, click "Save private key". Ignore the warning about having an empty passphrase, we will deal with that later. Save the private key to a file named `id_rsa.ppk`. Leave the PuTTYgen window open, as we will need to copy the public key shown at the top, in the field "Public key for pasting into OpenSSH authorized_keys file".

In order to login to a server, create a droplet in DigitalOcean. In the droplet creation form, at the "Add your SSH keys" section, click "New SSH Key". In the "SSH key content" field, paste the public key. In **Unix**, the public key is the contents of the file `id_rsa.pub`; in **Windows**, it is displayed at the top of the PuTTYgen window. When you create the droplet, it won't send you any email, as you won't need a password. The server will be ready for login with your SSH key.

Here is how to logon to the droplet from **Unix**:

```
ssh -i /tmp/id_rsa root@[server_ip_address]
```

In **Windows**, start PuTTY, and enter the server's IP address at the "Host Name" field (also look at Fig. 1.1). In addition, in the "Category" tree on the left, go to "Connection", "Data", and in "Auto-login username" enter "root"; then go to "SSH", "Auth", and in "Private key file for authentication" specify the `id_rsa.ppk` file; finally, go to "Session", specify a name in "Saved Sessions", and click "Save". Finally, click "Open". You should now login on the server without password.

What's more, in the future, if you just open PuTTY and double-click on the saved session name, you will immediately logon to the server.

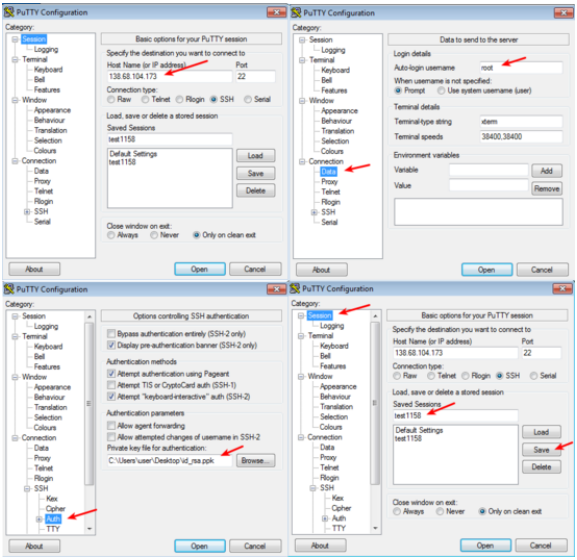


Fig. 1.1: How to configure PuTTY

1.4 How SSH keys work

As you noticed, the key generator created a public key and a private key. These "keys" are just numbers, integers, but large ones; if printed in decimal, they would be several hundreds of digits long. In order to save some space, they are stored in the files in a format that is more condensed than decimal, but the file format does not concern us (in fact, the private key file contains both keys, so if you lose the public key file you can generate it from the private key file using `ssh-keygen -y` in Unix or clicking "Load" on PuTTYgen).

These numbers are called keys because they are used in encryption and decryption. Encryption systems use keys. For example, a silly encryption system could be to replace a with b, b with c, and so on, so that the word "chair"

becomes “dibjs” and the word “zoo” becomes “app”. Or, instead of moving one letter forward you could move two letters forward, so “chair” becomes “ejckt” and “zoo” becomes “bqq”. In both cases, the algorithm is the same, but the key changes—in the first example the key is 1 (we moved one letter forward) and in the second it is 2 (two letters forward). In that algorithm, the key is a number from 1 to 25. If you send me an encrypted message with this algorithm and someone intercepts it, if they know the key with which it was encrypted they can decrypt it. Of course in this dummy system it’s trivial to find the key, and there are only 25 possible keys anyway, but what I want to illustrate here is that you need the key in order to decrypt the message. Serious encryption algorithms like AES are similar to our silly algorithm with respect to the fact that you decrypt with the same key that you used to encrypt, which gives them the name “symmetric”.

Now, asymmetric, or public key cryptography algorithms, such as RSA, have the property that keys go in pairs, and if you encrypt a message with one key, you can only decrypt it with the other key of the pair. What’s more, although there exists a method with which you can generate pairs of keys, if you know one of the two keys of a pair, you can’t derive the other. At least that’s what the mathematicians think. So our generator, `ssh-keygen` or `PuTTYgen`, generated a pair of two such numbers. It christened one of them “public key” and the other “private key”. So now if you want to send me an encrypted message I can just give you my public key, and it doesn’t matter if someone intercepts it. You can encrypt the secret message with my public key and send it to me, and it doesn’t matter if someone intercepts it. Only I have the corresponding private key, and only I can decrypt the message.

But how can this be used for authentication? Well, I can take any message, such as “hello world”, and encrypt it with my private key. I can then send it to you. You have my public key. You can decrypt the message. Since you were able to decrypt it with my public key, you know that it was encrypted with my private key. But only I have my private key, so it was I who encrypted the message. So you know I did it and no-one else. This is how digital signatures work, and how ssh authentication works.

So, the server is configured to accept login from you. It knows your public

key. The server asks the ssh client to encrypt some information with your private key. The ssh client (i.e. `ssh` on Unix or PuTTY on Windows) does so, and sends the encrypted information back to the server. The server verifies it can be decrypted with your public key, and then it gives you access.

You configure a server to accept SSH keys simply by adding them to `/root/.ssh/authorized_keys`, one public key per line. Logon to the server and examine the contents of the file (`nano` is the simplest text editor in GNU/Linux systems):

```
nano /root/.ssh/authorized_keys
```

You will see that it contains a line with the SSH key you pasted from `PuTTYgen` or from the `id_rsa.pub` file. That is all the Digital Ocean UI did, it just added the key to that file. You can specify many allowed keys in that file, one key per line. `/root` is the home directory of the root user, so the keys specified in `/root/.ssh/authorized_keys` may logon as root. If the system has a user named joe, the home directory of joe is usually `/home/joe`, and the keys allowed to logon as joe will be stored in `/home/joe/.ssh/authorized_keys`.

It's not only people who have SSH keys; SSH servers also have their own keys. Your server has key pairs in `/etc/ssh`. It's not only you who needs to authenticate with the server, but the server also needs to authenticate with you. You really need to know that you are logging in to your own server; an attacker could have compromised your local DNS cache and be directing you towards another server. They could steal valuable information if they did that, or obtain access to the real server. This is why, the first time you connect to a server, your SSH client gives you a warning. The server has provided its public key and has proven that it has the corresponding private key, but the SSH client has never connected to this server before, so it has no way of verifying that that server is really that server. The next times you connect to that server there will be no warning, because the SSH client can now verify that it is the same server as the server it connected to the previous time. On Unix, the ssh client stores server keys in `$HOME/.ssh/known_hosts`; PuTTY stores them in the registry, in `HKEY_CURRENT_USER\Software\SimonTatham\PuTTY\SshHostKeys`.

You may have noticed that the warning mentions the "fingerprint" of the key of the server. It could have just given us the public key, but this would have been inconvenient because keys are so large. Therefore to identify keys we use hashes of the keys which we call fingerprints and are much smaller; they are usually numbers with 32 hexadecimal digits. While it cannot be ruled out that two different keys might have the same fingerprint, the probability of this happening is lower than the sky falling on your head in the next minute, so it's not something you should worry about.

1.5 Using an SSH agent

Usually you only need a single pair of keys. On **Unix**, we usually store them at `$HOME/.ssh/id_rsa` and `$HOME/.ssh/id_rsa.pub`. `ssh-keygen` by default places them there, and the ssh client, "ssh", uses them without needing to specify any parameter. On **Windows** there is no prescribed location, so you should put your `.ppk` file wherever it is convenient.

Now, whoever has your private key can probably format all your servers, and possibly delete all your software on GitHub. If your laptop is stolen, they get your key (unless your disk is encrypted). It doesn't matter if they don't know your boot password or your login password. Anyone with a screwdriver can reset your BIOS password, and there are several ways to access a disk when you don't know the login password; one of the most obvious is to plug the disk on another system. For this reason, you should encrypt your private key file with a passphrase. You can either create a new key and not give it an empty passphrase, or you can change the passphrase of an existing key. You do this with `ssh-keygen -p -f /tmp/id_rsa` (you can omit the `-f /tmp/id_rsa` part if you want to use the default file, `$HOME/.ssh/id_rsa`), or by loading the key in `PuTTYgen`, specifying a passphrase, and saving it again.

But it doesn't make any sense to key in the passphrase each time you want to login to the server. There would be little advantage over typing the password each time. So what we do is run an "agent", software that runs continuously in the background, and keeps our unencrypted private key cached in memory.

The ssh client communicates with the agent whenever needed and gets the key from there. The agent only asks for the passphrase once after you login to your local machine, and then keeps it cached until logout or shutdown. This, combined with a screen saver that locks your screen after a few minutes of inactivity (I use 5 minutes), is reasonably secure.

On **GNU/Linux**, you don't need to do anything. `ssh-agent`, as the agent is called, is installed by default. The first time you attempt to ssh into the remote server, it will ask you for your passphrase.

On **Windows**, you need to download `pageant.exe` from the PuTTY download page and set it up to start at login. On Windows 7, you go to Start, All programs, Startup folder, right-click on the folder and select "Open", and in there create a new shortcut which should execute `C:\...\pageant.exe C:\...\your_key.ppk`. After you do that, try to logout and login (or restart the system altogether), and as soon as you login pageant will start and ask you for your passphrase.

Finally, on **Mac OS X**, I don't know how it works, but if you search the web for "Mac OS X ssh-agent" you should find enough information.

1.6 Essential GNU/Linux commands

Right after you login, enter this command:

```
pwd
```

This prints the working directory (also called the current directory), which right after login is `/root`, which is equivalent, very roughly, to `C:\Users\administrator` in Windows (which in older Windows versions was `C:\Documents¬and¬Settings\administrator`). `/root` is called the "home directory" of the root user. Most other users will have home directories under `/home`; for example, if there is user named joe, the home directory will usually be `/home/joe`; the root user is an exception.

In Unix-like operating systems, there is nothing like the drive letters of Windows. I just plugged a USB storage device on my Debian laptop, and I can see its files under `/media/anthony/ANTONIS`. Different storage devices are thus "mounted" in different locations of the single directory tree.

Now let's try to view the contents of the directory:

```
ls
```

"ls" stands for "list" and is the equivalent of the Windows `dir` command. If it didn't show anything, it's because the directory is empty. If you type `ls` on its own, it shows the contents of the current directory. Try listing the root directory instead:

```
ls /
```

You can make it list details by adding the `-l` parameter:

```
ls -l /
```

In that case, the output is like Fig. 1.2.

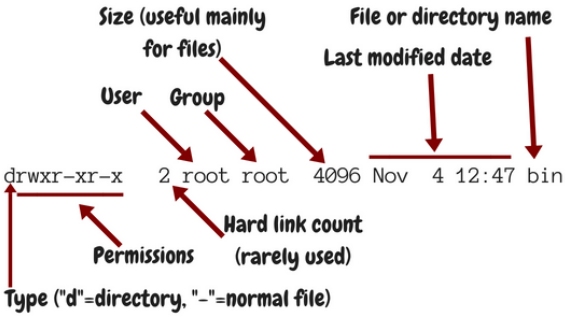


Fig. 1.2: Output of `ls`

Usually when we use `-l` we also use `-h`, which shows prettier numbers; for

example, instead of 4096 it shows 4.0K. You can type either `ls -l -h`, or, as is more common, `ls -lh`.

Just as in Windows, you can change directory using the `cd` command:

```
cd /
pwd
```

In contrast to Windows, `cd` on its own takes you to the home directory, so for the root user, a mere `cd` is equivalent to `cd /root`. The Unix-like equivalent of a mere `cd` in Windows is the `pwd` command.

Just a while ago, we tried `ls` on the `/root` directory. We are interested in some files that happen to be hidden. In Unix, when a file begins with a dot, it's "hidden". This means that `ls` doesn't normally show it, and that when you use wildcards such as * to denote all files, the shell will not include it. Otherwise it's not different from non-hidden files. To list the contents of a directory including hidden files, use the `-a` option:

```
ls -a
```

This will include `.` and `..`, which denote the directory itself and the parent directory (`/root/.` is the same as `/root`; `/root/..` is the same as `/`). You can use `-A` instead of `-a` to list all hidden files except `.` and `..`.

The last command we will examine in this section is `shutdown`. To restart a machine, enter `shutdown -r now`. You can also shut down a system with `shutdown -h now`, but this much is less often used on servers.

1.7 Shell files, editing files, remote copying

After the ssh server authenticates you and decides to give you access, it runs your shell. The shell is the program that accepts input from you, parses it, and executes the commands you type. There is a number of shells you can choose from, but most probably you are using the most popular, which is

called "bash". Bash stores the commands you type in `.bash_history`; when at the shell prompt you use the arrow up/down keys to move through your history of commands, bash gets this history from the file.

When you login, bash executes the commands in `.profile`; and when you logout, it executes the commands in `.bash_logout`. Finally, each time an interactive shell starts, it executes the commands in `.bashrc`. The difference between `.bashrc` and `.profile` is that the latter is executed only by a "login shell"; that is, by the shell started by the ssh server as soon as you login; but if you start another shell, e.g. by entering `bash`, only `.bashrc` is executed. Type this:

```
bash
exit
```

The first command starts another bash that runs inside the bash you were running before. The second command exits from the nested bash and returns you to the previous bash. Of course you would normally not do something like this, but it demonstrates that the "outside" shell is probably your login shell, whereas the "inside" shell is another interactive shell. When the nested one starts, it executes `.bashrc`.

Now, let's edit `.bashrc`.

```
nano .bashrc
```

We have already seen `nano` before—it's the simplest editor in GNU/Linux systems like Debian and Ubuntu. Many people prefer to use `vim` or `emacs`, which are very powerful but need some learning. `nano` is as simple as Windows Notepad, but it does not need a GUI. At the bottom it shows you what the special keys do; for example, ^X (Ctrl+X) exits the editor.

I like the bash prompt to be colored. To use the same colors I use, add this snippet at the end of the .bashrc file:

```
red=$(tput setaf 1)
green=$(tput setaf 2)
```

```
blue=$(tput setaf 4)
reset=$(tput sgr0)
PS1='\[$red\]\u\[$reset\]@\[$green\]\h\[$reset\]:\[$blue\]\W\[
 ↪$reset\]\$ '
```

Now exit nano by saving the file, logout and login again, and the prompt should be colored. I'm not going to explain how these commands work, as they are quite complicated; my main purpose here was for you to get a grip with editing a file and see the results.

If you have custom stuff in your `.bashrc`, you won't want to login to the server, edit `.bashrc`, make the changes, save, logout, and login again, and all that each time you create a new virtual server. Instead, you will want to keep your custom `.bashrc` somewhere in your local machine and copy it to the new virtual server. If your local machine runs GNU/Linux, you can use the `scp` command:

```
scp .bashrc root@1.2.3.4:/root/
```

If you have Windows, download `pscp.exe` from the PuTTY download page, make sure it's in the system path, and run it from a command prompt or PowerShell like this:

```
pscp .bashrc root@1.2.3.4:/root/
```

The command means "open an ssh connection to machine 1.2.3.4, login as root, and using the ssh connection transfer the file `.bashrc` from the local machine to the remote, and put it in `/root/`". It uses the ssh keys stored in `.ssh` or in PuTTY, so it can login without a password. Instead of `.bashrc` you could have used a full or relative path such as `/home/anthony/.bashrc` or `C:\Users\user\.bashrc`. Instead of `root@1.2.3.4:/root/` you could have simply used `root@1.2.3.4:` (don't forget the colon at the end); if you don't specify a destination, the remote user's home directory is the default. Copying also works the other way round; `scp root@1.2.3.4:.bashrc .` would fetch the remote file `/root/.bashrc` locally and put it in the current directory.

1.8 Installing software on a Debian/Ubuntu server

If you want to install nginx or any other software on Windows, you need to go to the software's web site, download it, and execute the downloaded installer. In Debian and Ubuntu we rarely do something like this. To install nginx, just enter this command:

```
apt install nginx
```

`apt` is the Advanced Package Tool. Except for "install" it also has "remove", with which you can uninstall, and some other options. You will find out that people mostly use `apt-get` instead of `apt`, which is also correct. `apt` actually uses `apt-get` behind the scenes.

What actually happens is that the Debian/Ubuntu developers have packaged nginx so that it can be installed with `apt`. They have done this with thousands of software items, so whenever you want to install something on your server, chances are it's packaged. This is true for other GNU/Linux systems as well, though they usually use different package managers. CentOS uses `rpm`, for example.

`apt` keeps a list of available packages. This needs to be updated regularly, because it changes whenever there are security updates. Try this to update the list:

```
apt update
```

After you update the list, you also need to upgrade any installed packages:

```
apt upgrade
```

After creating a new server, pretty much the first thing you should do is to update the list and upgrade the software. The two commands can be joined into a single one like this:

```
apt update && apt -y upgrade
```

This idiom means "run `apt update`, and if it succeeds, run `apt -y upgrade`". The `-y` option tells apt to not ask you "Do you want to continue?", but instead assume yes.

In contrast to Windows, which installs updates automatically, Debian/Ubuntu servers don't, unless you install package `unattended-upgrades`. However, I don't recommend it. I think it's a bad idea to run unattended upgrades, and I've once seen a server stop working when it was performing unattended upgrades and the upgrade procedure needed to ask a question. We found out the next morning. What I do is that I am subscribed to the Debian Security Announce mailing list, so whenever there's a problem I get notified and I run `apt update && apt -y upgrade`. I'm using Debian on all my servers; if you use Ubuntu, you should subscribe to the Ubuntu Security Announce list instead.

This applies only to software installed with `apt`. If you install software in any other way, `apt` will not upgrade it, and the Debian/Ubuntu security announce mailing lists will not mention it. The most common other way that you will use to install software is `pip`. You will probably install Django with `pip`, and you should be monitoring the Django blog for security announcements (you can subscribe to its feed, for example).

Sometimes you will not know how a package is named. Suppose you want to install Apache. You immediately suspect that Apache may be packaged, but you don't know the name of the package. Here is how to search for Apache:

```
apt-cache search apache
```

(`apt search apache` also works, but `apt-cache search` is faster and I like better the formatting of the results.) On a Ubuntu 16.04 system, this returns about 735 results. If you only want to search for packages that have "apache" in their name (and not just in their description or elsewhere), you can do this:

```
apt-cache search --names-only apache
```

This returns 161. Still many. You can narrow it down by searching only for packages whose name begins with "apache":

```
apt-cache search --names-only ^apache
```

This returns only 12 packages. The first one, `apache2`, is probably what you want. You can examine the contents of the package thus:

```
apt show apache2
```

There are more ways to narrow down the search, as there are tens of thousands of packages, but I think that's enough for now.

1.9 Reading the documentation

In the preceding sections, we saw that `ls` can accept several options, such as `-l`, `-h`, `-a`, `-A`, and others, and that `apt-cache search` accepts the `--names-only` option. Where can you find a reference of the options used?

The answer depends on the tool. Traditionally we use the `man` command for this; for example,

```
man apt-cache
```

will show you the full documentation of `apt-cache`.

In 1990 we were still reading the documentation from printed manuals, and `man ls` would show you the contents of the printed manual's `ls` entry. "man" stands for manual. At that time, that system was quite cool. If you wanted to take a quick glance at a detail in the manual you'd use the `man` command which was quicker, but if you wanted to study the manual more carefully you'd prefer to use the printed version which was easier to read. Remember,

there was no web at that time, and terminals weren't as smart as they are today (there was no bold or italics when you used the `man` command).

When the GNU system was developed at around that time, its developers thought that the `man` system was outdated, and they developed `info`. Although this is a better system that uses hyperlinks, it didn't get much traction, so today it's not much used. You can access the full documentation for `ls` with `info ls`, but this works much better from within the emacs editor than with the standalone `info` program, and it takes some learning. I never use `info`; I usually just use `man ls`, which is a summary that has most of the information I need, and if I need more I usually search the web.

Finally, it has lately become fashionable for commands to show help when given the `--help` option. Usually the help provided with `--help` is more condensed than that provided by `man` or `info`. `ls` has all three; `info`, `man`, and `--help`.

The quality of the documentation varies. While sometimes the help provided by `man` is excellent and can be used as tutorial as well as reference, very often it is better to familiarize yourself with a program by reading a book or a tutorial on the web. For example, you can't possibly learn `git` from its official documentation (and you can barely use it as reference).

1.10 Setting up the system locale

The "locale" is the regional settings, among which the character encoding used. If the character encoding isn't correctly set to UTF-8, sooner or later you will run into problems. So checking the system locale is pretty much the first thing you should do on a new server.

The procedure is this:

1. Open the file `/etc/locale.gen` in an editor and make sure the line that begins with "en_US.UTF-8" is uncommented.
2. Enter the command `locale-gen`; this will (re)generate the locales.

3. Open the file `/etc/default/locale` in an editor, and make sure it contains the line `LANG=en_US.UTF-8`. Changes in this file require logout and login to take effect.

Let me now explain what all this is about. The locale consists of a language, a country, and a character encoding; "en_US.UTF-8" means English, United States, UTF-8. This tells programs to show messages in American English; to format items such as dates in the way it's done in the United States; and to use encoding UTF-8.

Different users can be using different locales. If you have a desktop computer used by you and your spouse, one could be using English and the other French. Each user does this by setting the `LANG` environment variable to the desired locale; if not, the default system locale is used for that user. For servers this feature is less important. While your Django application may display the user interface in different languages (and format dates and numbers in different ways), this is done by Django itself using Django's internationalization and localization machinery and has nothing to do with what we are discussing here, which affects mostly the programs you type in the command line, such as `ls`. Because for servers the feature of users specifying their preferred locale isn't so important, we usually merely use the default system locale, which is specified in the file `/etc/default/locale`. You can understand English, otherwise you wouldn't be reading this book, so "en_US.UTF-8" is fine. If you prefer to use another country, such as "en_UK.UTF-8", it's also fine, but it's no big deal, as I will explain later on.

Although the system can support a large number of locales, many of these are turned off in order to save a little disk space. You turn them on by adding or uncommenting them in file `/etc/locale.gen`. When you execute the program `locale-gen`, it reads `/etc/locale.gen` and determines which locales are activated, and it compiles these locales from their source files, which are relatively small, to some binary files that are those actually used by the various programs. We say that the locales are "generated". If you activate all locales the binary files will be a little bit over 100 M, so the saving is not that big (it was important 15 years ago); however they will take quite some time to generate. Usually we only activate a few.

To check that everything is right, do this:

1. Enter the command `locale`; everything (except, possibly, `LANGUAGE` and `LC_ALL`) should have the value "en_US.UTF-8".
2. Enter the command `perl -e ''`; it should do nothing and give no message.

The `locale` command merely lists the active locale parameters. `LC_CTYPE`, `LC_NUMERIC` etc. are called "locale categories", and usually they are all set to the same value. In some edge cases they might be set to different values; for example, on my laptop I use "en_US.UTF-8", but especially for `LC_TIME` I use "en_DK.UTF-8", which causes Thunderbird to display dates in ISO 8601. This is not our concern here and it rarely is on a server. So we don't set any of these variables, and they all get their value from `LANG`, which is set by `/etc/default/locale`.

However, sometimes you might make an error; you might specify a locale in `/etc/default/locale`, but you might forget to generate it. In that case, the `locale` command will indicate that the locale is active, but it will not show that anything is wrong. This is the reason I run `perl -e ''`. Perl is a programming language, like Python. The command `perl -e ''`, does nothing; it tells Perl to execute an empty program; same thing as `python -c ''`. However, if there is anything wrong with the locale, Perl throws a big warning message; so `perl -e ''` is my favourite way of verifying that my locale works. Try, for example, `LANG=el_GR.UTF-8 perl -e ''` to see the warning message. So `locale` shows you which is the active locale, and `perl -e ''`, if silent, indicates that the active locale has been generated and is valid.

I told you a short while ago that the country doesn't matter much for servers. Neither does the language. What matters is the encoding. You want to be able to manipulate all characters of all languages. Even if all your customers are English speaking, there may eventually be some remark about a Chinese character in a description field. Even if you are certain there won't, it doesn't make any sense to constrain yourself to an encoding that can represent only a subset of characters when it's equally easy to use UTF-8. So you need to make sure you use UTF-8. In the chapter about PostgreSQL we will see that

installing PostgreSQL is a process particularly sensitive to the system locale settings.

The programs you run at the command line will be producing output in your chosen encoding. Your terminal reads the bytes produced by these programs and must be able to decode them properly, so it must know how they are encoded. In other words, you must set your terminal to UTF-8 as well. Most terminals, including PuTTY and gnome-terminal, are by default set to UTF-8, but you can change that in their preferences.

1.11 Quickly starting Django on a server

As I said in the beginning, we will be experimenting. Experimenting means we will be trying things. We will be installing your Django project and do things with it, and then we will be deleting it and reinstalling it to try things differently as we move on. You must have mastered setting up a development server from scratch. You should be able to setup your Django project on a newly installed machine within a couple of minutes at most, with a sequence of commands similar to the following:

```
apt install git python3 virtualenvwrapper
git clone $DJANGO_PROJECT_REPOSITORY
cd $DJANGO_PROJECT
mkvirtualenv --system-site-packages $DJANGO_PROJECT
pip install -r requirements.txt
python3 manage.py migrate
python3 manage.py runserver
```

It doesn't matter if you use Python 2 instead of 3, or `mercurial` (or even, horrors, FTP) instead of `git`, or plain `virtualenv` instead of virtualenvwrapper, or if you don't use `--system-site-packages`. What *is* important is that you have a grip on a sequence of commands similar to the above and get your development server running in one minute. We will be using `virtualenv` heavily; if you aren't comfortable with `virtualenv`, read my blog post on

virtualenv.

So, you have your virtual server, and you have a sequence of commands that can install a Django development server for your project. Go ahead and do so on the virtual server. Do it as the root user, in the `/root` directory.

Now, make sure you have this in your settings:

```
DEBUG = True
ALLOWED_HOSTS = ['$SERVER_IPv4_ADDRESS']
```

Then, instead of running the development server with `./manage.py runserver` run it as follows:

```
./manage.py runserver 0.0.0.0:8000
```

After it starts, go to your web browser and tell it to go to http://$SERVER_IPv4_ADDRESS:8000/. You should see your Django project in action.

Usually you run the Django development server with `./manage.py runserver`, which is short for `./manage.py runserver 8000`. This tells the Django development server to listen for connections on port 8000. However, if you just specify "8000", it only listens for local connections; a web browser running on the server machine itself will be able to access the Django development server at "http://localhost:8000/", but remote connections, from another machine, won't work. We use "0.0.0.0:8000" instead, which asks the Django development server to listen for remote network connections. Even better, if your virtual server has IPv6 enabled, you can use this:

```
./manage.py runserver [::]:8000
```

This will cause Django to listen for remote connections on port 8000, both for IPv4 and IPv6.

Next problem is that you can't possibly ask your users to use http://$SERVER_IPv4_ADDRESS:8000/. You have to use a domain name,

and, you have to get rid of the ":8000" part. Let's deal with the ":8000" first. "http://$SERVER_IPv4_ADDRESS/" is actually a synonym for "http://$SERVER_IPv4_ADDRESS:80/", so we need to tell Django to listen on port 80 instead of 8000. This may or may not work:

```
./manage.py runserver 0.0.0.0:80
```

Port 80 is privileged. This means that normal users aren't allowed to listen for connections on port 80; only the root user is. So if you run the above command as as a normal user, Django will probably tell you that you don't have permission to access that port. If you run the above command as root, it should work. If it tells you that the port is already in use, it probably means that a web server such as Apache or nginx is already running on the machine. Shut it down:

```
service apache2 stop
service nginx stop
```

When you finally get `./manage.py runserver 0.0.0.0:80` running, you should, at last, be able to go to your web browser and reach your Django project via http://$SERVER_IPv4_ADDRESS/. Congratulations!

1.12 Things we need to fix

Now, of course, this is the wrong way to do it. It's wrong for the following reasons:

- The URL http://$SERVER_IPv4_ADDRESS/ is ugly; you need to use a domain name.
- You have put your project in `/root`.
- You are running Django as root.
- You have Django serve your static files, and you have DEBUG=True.

- You are using `runserver`, which is seriously suboptimal and only meant for development.
- You are using SQLite.

Let's go fix them.

CHAPTER

TWO

DNS

2.1 Introduction to the DNS

In this book, you will find that I like to show you the code first, even if you don't understand it clearly, and then explain to you how things work. Unfortunately, I cannot do that with DNS. You need to understand it first and then write the code. **The big problem with DNS is that if you screw things up, even if you fix or revert things, it may be days before the system works again**. So you need to read carefully.

When you open your browser and type http://djangodeployment.com/, the first thing your browser does is find the IP address of the machine djangodeployment.com. For this, it asks a component of the operating system called the "resolver": "What is the IP address of djangodeployment.com?" After some time (usually from a few ms to a few seconds), the resolver replies: "It's 71.19.145.109". The browser then proceeds to open a TCP connection on port 80 of that address and use HTTP to request the required information (in our case the home page of djangodeployment.com).

Note: What about IPv6?

If your computer has an IPv6 connection to the Internet, your browser will actually first ask the resolver for the IPv6 address of server. For djangodeployment.com, the resolver will eventually reply "It's 2605:2700:0:3::4713:916d". The browser will then attempt to connect to that IPv6 address. If there is any kind of error, such as the resolver being unable to find an IPv6 address (many web servers aren't yet configured to use one), or the IPv6 address not responding (network errors are still more frequent with IPv6 than IPv4), the browser will fall back to using the IPv4 address, as I explained above.

The only thing the resolver does is ask another machine to do the actual resolving; that other machine is called a name server. Most likely you are using a name server provided by your Internet Service Provider. I will be calling that name server "your name server", although it's not exactly yours; but it's the one you are using.

Tip: Which is my name server?

On Unix-like machines (including Mac OS X), the name server used is stored in file `/etc/resolv.conf`; the file is usually setup during DHCP, but on systems with a static IP address it is often edited manually. On Windows, you can determine the name server by typing the command 'ipconfig /all', where it shows as "DNS Servers"; it is setup during DHCP, but on systems with a static IP address it is often edited manually in the network properties. Your system may be configured to use more than one name server, in which case it chooses one and uses another if the first one does not respond.

You might find out that the name server is your aDSL router. Actually your aDSL router is merely a so-called "forwarding" name server, which only transfers the query to another name server, which is the one that does the real magic. You can find which one it is by logging in your router's web interface and browsing through its settings. It is setup during the

establishment of the aDSL connection.

When I say "your name server" I don't mean the forwarding name server, but the one that does the real job.

In order to find out the address that corresponds to a name, your name server makes a series of questions to other name servers on the Internet:

1. First, your name server picks up one of thirteen so-called "root name servers". The IP addresses of these thirteen name servers are well-known (the official list is at http://www.internic.net/domain/named.root) and generally do not change, and your name server is preprogrammed to use them. Your name server tells the chosen root name server something like this: "Hello, I'd like to know the IP address of djangodeployment.com please."
2. The root name server replies: "Hi. I don't know the address of djangodeployment.com; you should ask one of these name servers, which are responsible for all domain names ending in '.com"' (and it supplies a number of IP addresses (actually thirteen).
3. Your name server picks up one of the .com name servers and asks it: "Hello, I'd like to know the IP address of djangodeployment.com please."
4. The .com name server replies: "Hi. I don't know the address of djangodeployment.com; you should ask one of these name servers, which are responsible for djangodeployment.com" (and it supplies a number of IP addresses, which at the time of this writing are three).
5. Your name server picks up one of the three name servers and asks it: "Hello, I'd like to know the IP address of djangodeployment.com please."
6. The djangodeployment.com name server replies: "Sure, djangodeployment.com is 71.19.145.109".

After your name server gets this information, it replies to the resolver, which in turn replies to your browser.

In this example, there were only six steps, but they could be more; for example, if you try to resolve cs.man.ac.uk, first the root servers will be asked, these will direct to the .uk name servers, which will direct to the .ac.uk name servers, and so on, for a total of 10 steps (this is not always the case; when resolving itia.civil.ntua.gr, the .gr servers refer you to the .ntua.gr servers, and these in turn refer you directly to the itia.civil.ntua.gr servers, for a total of 8 steps).

All this discussion between servers takes time and network traffic, so it only happens the first time you ask to connect to the web page. The results of the DNS query are heavily cached in order to make it faster for the next times. Typically web browsers cache such results for about half an hour, or until browser restart. Most important, however, your name server caches results for much longer. In fact, the response (6) above is not exactly what I wrote; instead, it is "Sure, djangodeployment is 71.19.145.109, and you can cache this information for up to 8 hours". Equally important, the response (4) is "I don't know the address of djangodeployment.com; you should ask one of these three name servers, which are responsible for djangodeployment.com, and you can cache this information (i.e. the list of name servers that are responsible for djangodeployment.com) for up to two days". Caching times are configurable to various degrees and are usually from 5 minutes to 48 hours, but caching for a whole week is not uncommon. Rarely does your name server need to go through the complete list of steps; most often it will have cached the name servers for the top level domain, and sometimes it will also have cached some lower stuff.

So here is the big problem with DNS: it's not hard to get it right (it's easier than writing a Django program), but if you make the slightest error you might be stuck with the wrong information for up to two days (or even a week). If you make an error when configuring your domain name, and a customer attempts to access your site, the error may be cached by the customer's name server for up to two days, and you can do nothing about it except fix the error and wait. There is no way to send a signal to all the name servers of the world

and tell them "hey, please invalidate the cache for djangodeployment.com". Different customers or visitors of your site will experience different amounts of downtime, depending on when exactly their local name server will decide to expire its cache.

2.2 Registering a domain name

You register a domain name with a registrar. Registrars are companies that provide the service of registering a domain name for you. These companies are authorized by ICANN, the organization ultimately responsible for domain names. So, before registering a domain name, you first need to select a registrar, and there are many. I'm using BookMyName.com, a French registrar which I selected more or less at random. Its web site is unpolished but it works. Another French registrar, particularly popular in the free software community, is Gandi, but it's a bit more expensive than others. The most popular registrar worldwide is GoDaddy, but it supported SOPA, and for me that's a deal breaker. Another interesting option is Namecheap; I think its software is nice and its prices are reasonable. If you don't know what to do, choose that one. There are also dozens of other options, and it's fine to choose another one. Note that I'm not affiliated with any registrar (and certainly none of the four I've mentioned).

For practice, you can go and register a cheap test domain; Namecheap, for example, sells some domains for $0.88 per year. Go get one now so that you can start messing around with it. Below I use ".com" as an example, but if your domain is different ($0.88 domains certainly aren't .com) it doesn't matter, exactly the same rules apply.

When you register a .com domain name at the registrar's web site, two things happen:

1. The registrar configures some name servers to be the name servers for the domain. For example, when I registered djangodeployment.com at the web site of bookmyname.com, bookmyname.com configured three

name servers (nsa.bookmyname.com, nsb.bookmyname.com, and nsc.bookmyname.com) as the djangodeployment.com name servers. These are the three servers that are involved in steps 5 and 6 of the resolving procedure that I presented in the previous section. I am going to call them the **domain's name servers**.

2. The registrar notifies the .com name servers that domain djangodeployment.com is registered, and that the site name servers are the three mentioned above. I am going to call the .com name servers the **upstream name servers**. If your domain is mydomain.co.uk, the upstream name servers are those responsible for .co.uk.

2.3 Adding records to your domain

The DNS database consists of records. Each record maps a name to a value. For example, a record says that the name djangodeployment.com corresponds to the value 71.19.145.109. Your registrar provides a web interface with which you can add, remove and edit records (in Namecheap you need to go to the Dashboard, Domain list, Manage (the domain), Advanced DNS). Go to your registrar's interface and, for the test domain you created, create the following records (remember that $SERVER_IPv4_ADDRESS and $SERVER_IPv6_ADDRESS are placeholders and you need to replace them with something else; also omit the "AAAA" records if your server doesn't have an IPv6 address):

Name	Type	TTL	Value
@	A	300	$SERVER_IPv4_ADDRESS
@	AAAA	300	$SERVER_IPv6_ADDRESS
www	A	300	$SERVER_IPv4_ADDRESS
www	AAAA	300	$SERVER_IPv6_ADDRESS

Each record has a type. There are many different types of records, but the ones you need to be aware of here are A, AAAA, and CNAME. "A" defines an IPv4 address, whereas "AAAA" defines an IPv6 address. We will deal with

CNAME a bit later.

When you see "@" as a name, I mean a literal "@" symbol. This is shorthand for writing the domain itself. If your domain is mydomain.com, then whether you enter "mydomain.com." (with a trailing dot) or "@" in the field for the name is exactly the same thing. Some registrars might be allowing only the shorthand "@", but often it is allowed to write "mydomain.com.". Use the "@", which is more common. The first of these four records means that the domain itself resolves to $SERVER_IPv4_ADDRESS. Likewise for the second record.

If your domain is mydomain.com, the next two records define the IP addresses for www.mydomain.com. In the field for the name, you can either write "www.mydomain.com." (with a trailing dot), or "www", without a trailing dot. Use the latter, which is more common. In the rest of the text, I will be using $DOMAIN and www.$DOMAIN instead of mydomain.com and www.mydomain.com, and you should understand that you need to replace "$DOMAIN" with your actual domain.

These four records are normally all you need to set. In theory you can set www.$DOMAIN to point to a different server than $DOMAIN, but this is uncommon. You can also define ftp.$DOMAIN and whateverelse.$DOMAIN, but this is often not needed.

The TTL, meaning "time to live", is the maximum allowed caching time. When a name server asks the domain's name server for the IPv4 address of $DOMAIN, the domain's name server will reply "$DOMAIN is 71.19.145.109, and you can cache this information for 300 seconds". Don't make it less than 300; it will increase the number of queries your visitors will make, thus making responses a bit slower; and some name servers will ignore the TTL if it's less than 300 and use 300 anyway. A common tactic is to use a large value (say 28800), and when for some reason you need to switch to another server, you reduce that to 300, wait at least 8 hours (28800 seconds), then bring the server down, change the DNS to point to the new server, then start the new server. If planned correctly and executed without problems, the switch will result in a downtime of no more than 300 seconds. After this is

finished, you change the TTL to 28800 again.

You can usually leave the TTL field empty. In that case, a default TTL applies. The default TTL for the zone ("zone" is more or less the same as a domain) is normally configurable, but this may depend on the web interface of the registrar.

CNAME records are a kind of alias. For example, one of the domains I'm managing is openmeteo.org, and its database is like this:

Name	Type	TTL	Value
@	A	300	83.212.168.232
@	AAAA	300	2001:648:2ffc:1014:a800:ff:feb1:6047
www	CNAME	300	ilissos.openmeteo.org.
ilissos	A	300	83.212.168.232
ilissos	AAAA	300	2001:648:2ffc:1014:a800:ff:feb1:6047

The machine that hosts the web service for openmeteo.org is called ilissos.openmeteo.org. When the name server is queried for www.openmeteo.org, it replies: "Hi, www.openmeteo.org is an alias; the canonical name is ilissos.openmeteo.org." So then it has to be queried again for ilissos.openmeteo.org. (However, you cannot use CNAME for the domain itself, only for other hosts within the domain.) On the right hand side of CNAMEs, you should always specify the fully qualified domain name **and end it with a dot**, such as "ilissos.openmeteo.org.", as in the example above.

I used to use CNAMEs a lot, but now I avoid them, because they make first-time visits a little slower. Assume you want to visit "http://www.openmeteo.org/synoptic/irma". Then these things happen:

1. www.openmeteo.org is resolved, and it turns out to be an alias of ilissos.openmeteo.org.
2. ilissos.openmeteo.org is resolved to an IP address.
3. The request http://www.openmeteo.org/synoptic/irma is sent to the IP address. The web server redirects it to http://openmeteo.org/synoptic/irma, without the www.

4. The request http://openmeteo.org/synoptic/irma is sent to the IP address, and it is redirected to http://openmeteo.org/synoptic/irma/, because I'm using `APPEND_SLASH = True` in Django's settings.

5. The request http://openmeteo.org/synoptic/irma/ is sent to the IP address, and this time a proper response is returned.

All these steps take a small amount of time which may add up to one second or more. This is only for the first request of first time visitors, but today people have little patience, and it's a good idea for the visitor's browser to start drawing something on the screen within at most one second, otherwise you will be losing a non-negligible number of visitors. Besides, a high quality web site should not have unnecessary delays. So lately I've stopped using CNAMEs, and I've stopped redirecting between URLs with and without the leading www.

2.4 Changing the domain's name servers

As I said, when you register the domain, the registrar configures its own name servers to act as the domain's name servers, and also tells the upstream name servers the ip addresses and/or names of the domain's name servers. While this is normally sufficient, there are cases when you will want to use other name servers instead of the registrar's name servers. For example, DigitalOcean offers name servers and a web interface to configure them, and if DigitalOcean's web interface is easier, or if it integrates well with droplets making configuration faster, you might want to use that. In such a case, you can go to the registrar's web interface and specify different name servers. The registrar will tell the upstream name servers which are your new name servers. It can't setup the new name servers themselves, you have to do that yourself (e.g. via the DigitalOcean's web interface if you are using DigitalOcean's name servers).

In this case, you must be aware that while, as we saw in the previous section, you can configure the TTL for the DNS records of your domain, **you cannot**

configure the TTL of the upstream name servers. The upstream name servers, when queried about your domain, respond with something like "the name servers for the requested domain are such and such, and you can cache this information for 2 days". This TTL, typically 2 days, is not configurable by you, so you have to live with it. So changing name servers is a bit risky, because if you do anything wrong, different users will experience different downtimes that can last for up to 2 days.

Finally, some information about the NS record, which means "name server". I haven't told you, but the DNS database (the zone file, as it is called) for djangodeployment.com also contains these records:

Name	Type	TTL	Value
@	NS	28800	nsa.bookmyname.com.
@	NS	28800	nsb.bookmyname.com.
@	NS	28800	nsc.bookmyname.com.

(As you can see, there can be many records with the same type and name, and this is true of A and AAAA records as well—one name may map to many IP addresses, but we will not delve into that here.)

I have never really understood the reason for the existence of these records **in the domain's zone file**. The upstream name servers obviously need to know that, but what's the use of querying a domain's name server about which are the domain's name servers? Obviously I already know them. However, there is a reason, and these records need to be present both in the domain's name servers and upstream.

In any case, these NS records are virtually always configured automatically by the registrar or by the web interface of the name server provider, so usually you don't need to know more about it. What you need to know, however, is that DNS is a complicated system that easily fills in several books by itself. It will work well if you are gentle with it. If you want to do something more advanced and you don't really know what you are doing, ask for help from an expert if you can't afford the downtime.

2.5 Editing the hosts file

As I told you earlier, when your browser needs to know the IP address that corresponds to a name, it asks your operating system's resolver, and the resolver asks the name server. It is possible to bypass the asking of the name server and tell the resolver what answers to give. This is done by modifying the `hosts` file, which in Unixes is `/etc/hosts`, and in Windows is `C:\Windows\System32\drivers\etc\hosts`. Edit the file and add these lines at the end:

```
1.2.3.4 mysite.com
1.2.3.4 www.mysite.com
```

Save the file, restart your browser (because, remember, it may be caching names), and then visit mysite.com. It will probably fail to connect (because 1.2.3.4 does not exist), but the thing is that mysite.com has resolved to 1.2.3.4. The resolver found it in the `hosts` file, so it did not ask the DNS server.

I often edit the `hosts` file, for experimenting with a temporary server without needing to change the DNS. Sometimes I want to redirect a domain to another machine, for development or testing, and I want to do this only for myself, without affecting the users of the domain. In such cases the `hosts` file comes in handy, and the changes made work immediately, without needing to wait for DNS caches to expire.

The only thing that you must take care of is to remember to revert the `hosts` file to its original contents; if you forget to do so, it might cause you great headaches later (imagine wondering why the web site you are deploying is different than what it should be, and discovering, after hours of searching, that it was because of a forgotten entry in `hosts`). What I usually do is leave the editor open and not close it until after I have reverted the file. When I don't do that thing, at least I make certain that the domain I'm playing with is `example.com` or anyway something very unlikely to ever be actually used by me.

2.6 Visiting your Django project through the domain

In the previous chapter you ran Django on a server and it was reachable through http://$SERVER_IPv4_ADDRESS/. Now you should have setup your DNS and have $DOMAIN point to $SERVER_IPv4_ADDRESS. In your Django settings, change `ALLOWED_HOSTS` to this:

```
ALLOWED_HOSTS = ['$DOMAIN', 'www.$DOMAIN']
```

Then run the Django development server as in the previous chapter:

```
./manage.py runserver 0.0.0.0:80
```

Now you should be able to reach your Django project via http://$DOMAIN/. So we fixed the first step; we managed to reach Django through a domain instead of an IP address. Next, we will run Django as an unprivileged user, and put its files in appropriate directories.

2.7 Chapter summary

- Register your domain at a registrar.
- Use the registrar's web interface to specify A and AAAA records for the domain and for www.
- Be careful when you play with TTLs and when changing the domain's name servers.
- If you do anything advanced with the DNS and you don't really know what you're doing and you can't afford the downtime, ask for expert help.
- Set `ALLOWED_HOSTS = ['$DOMAIN', 'www.$DOMAIN']`.

- Optionally use your local `hosts` file for experimentation.

CHAPTER

THREE

USERS AND DIRECTORIES

Right now your Django project is at `/root`, or maybe at `/home/joe`. The first thing we are going to fix is put your Django project in a proper place.

I will be using `$DJANGO_PROJECT` as the name of your Django project.

3.1 Creating a user and group

It's a good idea to not run Django as root. We will create a user specifically for that, and we will give the user the same name as the Django project, i.e. `$DJANGO_PROJECT`. However, in principle it can be different, and I will be using `$DJANGO_USER` to denote the user name, so that you can distinguish when I'm talking about the user and when about the project.

Execute this command:

```
adduser --system --home=/var/opt/$DJANGO_PROJECT \
    --no-create-home --disabled-password --group \
    --shell=/bin/bash $DJANGO_USER
```

Here is why we use these parameters:

`--system` This tells `adduser` to create a system user, as opposed to creating a normal user. System users are intended to run programs, whereas normal users are people. Because of this parameter, `adduser` will assign a user id less than 1000, which is only a convention for knowing that this is a system user. Otherwise there isn't much difference.

`--home=/var/opt/$DJANGO_PROJECT` This specifies the home directory for the user. For system users, it doesn't really matter which directory we will choose, but by convention we choose the one which holds the program's data. We will talk about the `/var/opt/$DJANGO_PROJECT` directory later.

`--no-create-home` We tell `adduser` to not create the home directory. We could allow it to create it, but we will create it ourselves later on, for instructive purposes.

`--disabled-password` The password will be, well, disabled. This means that you won't be able to become this user by using a password. However, the root user can always become another user (e.g. with `su`) without using a password, so we don't need one.

`--group` This tells `adduser` to not only add a new user, but to also add a new group, having the same name as the user, and make the new user a member of the new group. We will see further below why this is useful. I will be using `$DJANGO_GROUP` to denote the new group. In principle it could be different than `$DJANGO_USER` (but then the procedure of creating the user and the group would be slightly different), but the most important thing is that I want it to be perfectly clear when we are talking about the user and when we are talking about the group.

`--shell=/bin/bash` By default, `adduser` uses `/bin/false` as the shell for system users, which practically means they are disabled; `/bin/false` can't run any commands. We want the user to have the most common shell used in GNU/Linux systems, `/bin/bash`.

3.2 The program files

Your Django project should be structured either like this:

```
$DJANGO_PROJECT/
|-- manage.py
|-- requirements.txt
|-- your_django_app/
`-- $DJANGO_PROJECT/
```

or like this:

```
$REPOSITORY_ROOT/
|-- requirements.txt
`-- $DJANGO_PROJECT/
    |-- manage.py
    |-- your_django_app/
    `-- $DJANGO_PROJECT/
```

I prefer the former, but some people prefer the extra repository root directory.

We are going to place your project inside `/opt`. This is a standard directory for program files that are not part of the operating system. (The ones that are installed by the operating system go to `/usr`.) So, clone or otherwise copy your Django project in `/opt/$DJANGO_PROJECT` or in `/opt/$REPOSITORY_ROOT`. Do this **as the root user**. Create the virtualenv for your project **as the root user** as well:

```
virtualenv --system-site-packages --python=/usr/bin/python3 \
    /opt/$DJANGO_PROJECT/venv
/opt/$DJANGO_PROJECT/venv/bin/pip install \
    -r /opt/$DJANGO_PROJECT/requirements.txt
```

While it might seem strange that we are creating these as the root user instead of as `$DJANGO_USER`, it is standard practice for program files to belong to the root user. If you check, you will see that `/bin/ls` belongs to the root user,

though you may be running it as joe. In fact, it would be an error for it to belong to joe, because then joe would be able to modify it. So for security purposes it's better for program files to belong to root.

This poses a problem: when `$DJANGO_USER` attempts to execute your Django application, it will not have permission to write the compiled Python files in the `/opt/$DJANGO_PROJECT` directory, because this is owned by root. So we need to pre-compile these files as root:

```
/opt/$DJANGO_PROJECT/venv/bin/python -m compileall \
    -x /opt/$DJANGO_PROJECT/venv/ /opt/$DJANGO_PROJECT
```

The option `-x /opt/$DJANGO_PROJECT/venv/` tells compileall to exclude directory `/opt/$DJANGO_PROJECT/venv` from compilation. This is because the virtualenv takes care of its own compilation and we should not interfere.

3.3 The data directory

As I already hinted, our data directory is going to be `/var/opt/$DJANGO_PROJECT`. It is standard policy for programs installed in `/opt` to put their data in `/var/opt`. Most notably, we will store media files in there (in a later chapter). We will also store the SQLite file there. Usually in production we use a different RDBMS, but we will deal with this in a later chapter as well. So, let's now prepare the data directory:

```
mkdir -p /var/opt/$DJANGO_PROJECT
chown $DJANGO_USER /var/opt/$DJANGO_PROJECT
```

Besides creating the directory, we also changed its owner to `$DJANGO_USER`. This is necessary because Django will be needing to write data in that directory, and it will be running as that user, so it needs permission to do so.

3.4 The log directory

Later we will setup our Django project to write to log files in `/var/log/$DJANGO_PROJECT`. Let's prepare the directory.

```
mkdir -p /var/log/$DJANGO_PROJECT
chown $DJANGO_USER /var/log/$DJANGO_PROJECT
```

3.5 The production settings

Debian puts configuration files in `/etc`. More specifically, the configuration for programs that are installed in `/opt` is supposed to go to `/etc/opt`, which is what we will do:

```
mkdir /etc/opt/$DJANGO_PROJECT
```

For the time being this directory is going to have only `settings.py`; later it will have a bit more. Your `/etc/opt/$DJANGO_PROJECT/settings.py` file should be like this:

```
from DJANGO_PROJECT.settings import *

DEBUG = True
ALLOWED_HOSTS = ['$DOMAIN', 'www.$DOMAIN']
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': '/var/opt/$DJANGO_PROJECT/$DJANGO_PROJECT.db',
    }
}
```

Note: The above is not valid Python until you replace `$DJANGO_PROJECT` with the name of your django project and `$DOMAIN` with your domain. In all examples until now you might have been able to copy and paste the code from the book and use shell variables for `$DJANGO_PROJECT`, `$DJANGO_USER`, `$DJANGO_GROUP`, and so on. This is, indeed, the reason I chose this notation. However, in some places, like in this Python, you have to actually replace it yourself. (Occasionally I use DJANGO_PROJECT without the leading dollar sign, in order to get the syntax highlighter to work.)

Let's now **secure the production settings**. We don't want other users of the system to be able to read the file, because it contains sensitive information. Maybe not yet, but after a few chapters it is going to have the secret key, the password to the database, the password for the email server, etc. At this point, you are wondering: what other users? I am the only person using this server, and I have created no users. Indeed, now that it's so easy and cheap to get small servers and assign a single job to them, this detail is not as important as it used to be. However, it is still a good idea to harden things a little bit. Maybe a year later you will create a normal user account on that server as an unrelated convenience for a colleague.

If your Django project has a vulnerability, an attacker might be able to give commands to the system as the user as which the project runs (i.e. as `$DJANGO_USER`). Likewise, in the future you might install some other web application, and that other web application might have a vulnerability and could be attacked, and the attacker might be able to give commands as the user running that application. In that case, if we have secured our `settings.py`, the attacker won't be able to read it. Eventually servers get compromised, and we try to set up the system in such a way as to minimize the damage, and we can minimize it if we contain it, and we can contain it if the compromising of an application does not result in the compromising of other applications. This is why we want to run each application in its own user and its own group.

Here is how to make the contents of `/etc/opt/$DJANGO_PROJECT` unreadable

by other users:

```
chgrp $DJANGO_GROUP /etc/opt/$DJANGO_PROJECT
chmod u=rwx,g=rx,o= /etc/opt/$DJANGO_PROJECT
```

What this does is make the directory unreadable by users other than `root` and `$DJANGO_USER`. The directory is owned by `root`, and the first command above changes the group of the directory to `$DJANGO_GROUP`. The second command changes the permissions of the directory so that:

u=rwx The owner has permission to read (rx) and write (w) the directory (the `u` in `u=rwx` stands for "user", but actually it means the "user who owns the directory"). The owner is `root`. Reading a directory is denoted with `rx` rather than simply `r`, where the `x` stands for "search"; but giving a directory only one of the `r` and `x` permissions is an edge case that I've seen only once in my life. For practical purposes, when you want a directory to be readable, you must specify both `r` and `x`. (This applies only to directories; for files, the `x` is the permission to execute the file as a program.)

g=rx The group has permission to read the directory. More precisely, users who belong in that group have permission to read the directory. The directory's group is `$DJANGO_GROUP`. The only user in that group is `$DJANGO_USER`, so this adjustment applies only to that user.

o= Other users have no permission, they can't read or write to the directory.

You might have expected that it would have been easier to tell the system "I want `root` to be able to read and write, and `$DJANGO_USER` to be able to only read". Instead, we did something much more complicated: we made `$DJANGO_USER` belong to a `$DJANGO_GROUP`, and we made the directory readable by that group, thus indirectly readable by the user. The reason we did it this way is an accident of history. In Unix there has traditionally been no way to say "I want `root` to be able to read and write, and `$DJANGO_USER` to be able to only read". In many modern Unixes, including Linux, it is possible using Access Control Lists, but this is a feature added later, it does not work the same in all Unixes, and its syntax is harder to use. The way we use here

works the same in FreeBSD, HP-UX, and all other Unixes, and it is common practice everywhere.

Finally, we need to **compile** the settings file. Your settings file and the `/etc/opt/$DJANGO_PROJECT` directory is owned by root, and, as with the files in `/opt`, Django won't be able to write the compiled version, so we pre-compile it as root:

```
/opt/$DJANGO_PROJECT/venv/bin/python -m compileall \
    /etc/opt/$DJANGO_PROJECT
```

Compiled files are the reason we changed the permissions of the directory and not the permissions of `settings.py`. When Python writes the compiled files (which also contain the sensitive information), it does not give them the permissions we want, which means we'd need to be chgrping and chmoding each time we compile. By removing read permissions from the directory, we make sure that none of the files in the directory is readable; in Unix, in order to read file `/etc/opt/$DJANGO_PROJECT/settings.py`, you must have permission to read `/` (the root directory), `/etc`, `/etc/opt`, `/etc/opt/$DJANGO_PROJECT`, and `/etc/opt/$DJANGO_PROJECT/settings.py`.

You can check the permissions of a directory with the `-d` option of `ls`, like this:

```
ls -lhd /
ls -lhd /etc
ls -lhd /etc/opt
ls -lhd /etc/opt/$DJANGO_PROJECT
```

(In the above commands, if you don't use the `-d` option it will show the contents of the directory instead of the directory itself.)

Hint: Unix permissions

When you list a file or directory with the `-l` option of `ls`, it will show you something like `-rwxr-xr-x` at the beginning of the line. The first

character is the file type: - for a file and d for a directory (there are also some more types, but we won't bother with them). The next nine characters are the permissions: three for the user, three for the group, three for others. `rwxr-xr-x` means "the user has permission to read, write and search/execute, the group has permission to read and search/execute but not write, and so do others".

`rwxr-xr-x` can also be denoted as 755. If you substitute 0 in place of a hyphen and 1 in place of r, w and x, you get 111 101 101. In octal, this is 755. Instead of

```
chmod u=rwx,g=rx,o= /etc/opt/$DJANGO_PROJECT
```

you can type

```
chmod 750 /etc/opt/$DJANGO_PROJECT
```

which means exactly the same thing. People use this latter version much more than the other one, because it is so much easier to type, and because converting permissions into octal becomes second nature with a little practice.

3.6 Managing production vs. development settings

How to manage production vs. development settings seems to be an eternal question. Many people recommend, instead of a single `settings.py` file, a `settings` directory containing `__init__.py` and `base.py`. `base.py` is the base settings, those that are the same whether in production or development or testing. The directory often contains `local.py` (alternatively named `dev.py`), with common development settings, which might or might not be in the repository. There's often also `test.py`, settings that are used when testing. Both `local.py` and `test.py` start with this line:

```
from .base import *
```

Then they go on to override the base settings or add more settings. When the project is set up like this, `manage.py` is usually modified so that, by default, it uses `$DJANGO_PROJECT.settings.local` instead of simply `$DJANGO_PROJECT.settings`. For more information on this technique, see Section 5.2, "Using Multiple Settings Files", in the book Two Scoops of Django; there's also a stackoverflow answer about it.

Now, people who use this scheme sometimes also have `production.py` in the settings directory of the repository. Call me a perfectionist (with deadlines), but the production settings are the administrator's job, not the developer's, and your django project's repository is made by the developers. You might claim that you are both the developer and the administrator, since it's you who are developing the project and maintaining the deployment, but in this case you are assuming two roles, wearing a different hat each time. Production settings don't belong in the project repository any more than the nginx or PostgreSQL configuration does.

The proper place to store such settings is another repository—the deployment repository. It can be as simple as holding only the production `settings.py` (along with `README` and `.gitignore`), or as complicated as containing all your nginx, PostgreSQL, etc., configuration for several servers, along with the "recipe" for how to set them up, written with a configuration management system such as Ansible.

If you choose, however, to keep your production settings in your Django project repository, then your `/etc/opt/$DJANGO_PROJECT/settings.py` file shall eventually be a single line:

```
from $DJANGO_PROJECT.settings.production import *
```

However, I don't want you to do this now. We aren't yet going to use our real production settings, because we are going step by step. Instead, create the `/etc/opt/$DJANGO_PROJECT/settings.py` file as I explained in the previous section.

3.7 Running the Django server

Warning: We are running Django with `runserver` here, which is inappropriate for production. We are doing it only temporarily, so that you understand several concepts. We will run Django correctly in the chapter about *Gunicorn*.

```
su $DJANGO_USER
source /opt/$DJANGO_PROJECT/venv/bin/activate
export PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT
export DJANGO_SETTINGS_MODULE=settings
python /opt/$DJANGO_PROJECT/manage.py migrate
python /opt/$DJANGO_PROJECT/manage.py runserver 0.0.0.0:8000
```

You could also do that in an exceptionally long command (provided you have already done the `migrate` part), like this:

```
PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT \
    DJANGO_SETTINGS_MODULE=settings \
    su $DJANGO_USER -c \
    "/opt/$DJANGO_PROJECT/venv/bin/python \
    /opt/$DJANGO_PROJECT/manage.py runserver 0.0.0.0:8000"
```

Hint: su

You have probably heard of `sudo`, which is a very useful program on Unix client machines (desktops and laptops). On the server, `sudo` is less common and we use `su` instead.

`su`, like `sudo`, changes the user that executes a program. If you are user joe and you execute `su -c ls`, then `ls` is run as root. `su` will ask for the root password in order to proceed.

> `su alice -c ls` means "execute `ls` as user alice". `su alice` means "start a shell as user alice"; you can then type commands as user alice, and you can enter `exit` to "get out" of `su`, that is, to exit the shell than runs as alice. If you are a normal user `su` will ask you for alice's password. If you are root, it will become alice without questions. This should make clear how the `su` command works when you run the Django server as explained above.
>
> `sudo` works very differently from `su`. Instead of asking the password of the user you want to become, it asks for your password, and has a configuration file that describes which user is allowed to become what user and with what constraints. It is much more versatile. `su` does only what I described and nothing more. `su` is guaranteed to exist in all Unix systems, whereas `sudo` is an add-on that must be installed. By default it is usually installed on client machines, but not on servers. `su` is much more commonly used on servers and shell scripts than `sudo`.

Do you understand that very clearly? If not, here are some tips:

- Make sure you have a grip on virtualenv and environment variables.
- Python reads the `PYTHONPATH` environment variable and adds the specified directories to the Python path.
- Django reads the `DJANGO_SETTINGS_MODULE` environment variable. Because we have set it to "settings", Django will attempt to import `settings` instead of the default (the default is `$DJANGO_PROJECT.settings`, or maybe `$DJANGO_PROJECT.settings.local`).
- When Django attempts to import `settings`, Python looks in its path. Because `/etc/opt/$DJANGO_PROJECT` is listed first in `PYTHONPATH`, Python will first look there for `settings.py`, and it will find it there.
- Likewise, when at some point Django attempts to import `your_django_app`, Python will look in `/etc/opt/$DJANGO_PROJECT`;

it won't find it there, so then it will look in `/opt/$DJANGO_PROJECT`, since this is next in `PYTHONPATH`, and it will find it there.

- If, before running `manage.py [whatever]`, we had changed directory to `/opt/$DJANGO_PROJECT`, we wouldn't need to specify that directory in `PYTHONPATH`, because Python always adds the current directory to its path. This is why, in development, you just tell it `python manage.py [whatever]` and it finds your project. We prefer, however, to set the `PYTHONPATH` and not change directory; this way our setup will be clearer and more robust.

Instead of using `DJANGO_SETTINGS_MODULE`, you can also use the `--settings` parameter of `manage.py`:

```
PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT \
    su $DJANGO_USER -c \
    "/opt/$DJANGO_PROJECT/venv/bin/python \
    /opt/$DJANGO_PROJECT/manage.py
    runserver --settings=settings 0.0.0.0:8000"
```

(`manage.py` also supports a `--pythonpath` parameter which could be used instead of `PYTHONPATH`, however it seems that `--settings` doesn't work correctly together with `--pythonpath`, at least not in Django 1.8.)

If you fire up your browser and visit http://$DOMAIN:8000/, you should see your Django project in action.

3.8 Chapter summary

- Create a system user and group with the same name as your Django project.
- Put your Django project in `/opt`, with all files owned by root.
- Put your virtualenv in `/opt/$DJANGO_PROJECT/venv`, with all files owned by root.

- Put your data files in a subdirectory of `/var/opt` with the same name as your Django project, owned by the system user you created. If you are using SQLite, the database file will go in there.
- Put your settings file in a subdirectory of `/etc/opt` with the same name as your Django project, whose user is root, whose group is the system group you created, that is readable by the group and writeable by root, and whose contents belong to root.
- Precompile the files in `/opt/$DJANGO_PROJECT` and `/etc/opt/$DJANGO_PROJECT`.
- Run `manage.py` as the system user you created, after setting the environment variables `PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT` and `DJANGO_SETTINGS_MODULE=settings`.

CHAPTER

FOUR

THE WEB SERVER

This chapter is divided in two parts: nginx and Apache. Depending on which of the two you choose, you only need to read that part.

Both nginx and Apache are excellent choices for a web server. Most people deploying Django nowadays seem to be using nginx, so, if you aren't interested in learning more about what you should choose, pick up nginx. Apache is also widely used, and it is preferable in some cases. If you have any reason to prefer it, go ahead and use it.

If you want don't know what to do, choose nginx. If you want to know more about the pros and cons of each one, I have written a blog post about it.

4.1 Installing nginx

Install nginx like this:

```
apt install nginx-light
```

> **Note:** Instead of `nginx-light`, you can use packages `nginx-full` or `nginx-extras`, which have more modules available. However, `nginx-light` is enough in most cases.

After you install, go to your web browser and visit http://$DOMAIN/. You should see nginx's welcome page.

4.2 Configuring nginx to serve the domain

Create file `/etc/nginx/sites-available/$DOMAIN` with the following contents:

```
server {
    listen 80;
    listen [::]:80;
    server_name $DOMAIN www.$DOMAIN;
    root /var/www/$DOMAIN;
}
```

> **Note:** Again, this is not a valid nginx configuration file until you replace `$DOMAIN` with your actual domain name.

Create a symbolic link in `sites-enabled`:

```
cd /etc/nginx/sites-enabled
ln -s ../sites-available/$DOMAIN .
```

> **Hint:** Symbolic links

A symbolic link looks like a file, but in fact it is a pointer to another file. The command

```
ln -s ../sites-available/$DOMAIN .
```

means "create a symbolic link that points to file `../sites-available/$DOMAIN` and put the link in the current directory (`.`). Two dots denote the parent directory, so when the current directory is `/etc/nginx/sites-enabled`, `..` means the parent, `/etc/nginx`, whereas `../sites-available` means "one up, then down into `sites-available`. A single dot designates the current directory.

The command above is exactly equivalent as this:

```
ln -s ../sites-available/$DOMAIN $DOMAIN
```

which means "create a symbolic link that points to file `../sites-available/$DOMAIN` and give it the name $DOMAIN. If the last argument of `ln -s` is a directory (for example, `.`), then it creates the symbolic link in there and gives it the same name as the actual file.

You can treat the symbolic link as if it was a file; you can edit it with an editor, you can open it with a Python program using `open()`, and in these cases the actual file (the one being pointed to by the symbolic link) is opened instead.

While the order of arguments in the `ln` command may seem strange at first, it is consistent with the order of arguments in the `cp` command which merely copies files. Just as `cp source destination` copies file `source` to file `destination`, similarly `ln -s` is like making a copy of the file, but instead of an actual copy, it creates a symbolic link.

If you list files with `ls -l`, it is clearly indicated which file the symbolic link points to. The permissions of the link, `rwxrwxrwx`, may seem insecure, but they are actually irrelevant; it is the permissions of the actual file that count.

Except for symbolic links there are also hard links, which are created

> without the `-s` option, but are different and rarely used. It is unlikely that you will ever create a hard link, so get used to always type `ln -s`, that is, with the `-s` option.

Tell nginx to re-read its configuration:

```
service nginx reload
```

Finally, create directory `/var/www/$DOMAIN`, and inside that directory create a file `index.html` with the following contents:

```
<p>This is the web site for $DOMAIN.</p>
```

Fire up your browser and visit http://$DOMAIN/, and you should see the page you created.

The fact that we named the nginx configuration file (in `/etc/nginx/sites-available`) `$DOMAIN` is irrelevant; any name would have worked the same, but it's a convention to name it with the domain name. In fact, strictly speaking, we needn't even have created a separate file. The only configuration file nginx needs is `/etc/nginx/nginx.conf`. If you open that file, you will see that it contains, among others, the following line:

```
include /etc/nginx/sites-enabled/*;
```

So what it does is read all files in that directory and process them as if their contents had been inserted in that point of `/etc/nginx/nginx.conf`.

As we noticed, if you visit http://$DOMAIN/, you see the page you created. If, however, you visit http://$SERVER_IPv4_ADDRESS/, you should see nginx's welcome page. If the host name (the part between "http://" and the next slash) is $DOMAIN or www.$DOMAIN then nginx uses the configuration we specified above, because of the `server_name` configuration directive which contains these two names. If we use another domain name, or the server's ip address, there is no matching `server { ... }` block in the nginx config-

uration, so nginx uses its default configuration. That default configuration is in `/etc/nginx/sites-enabled/default`. What makes it the default is the `default_server` parameter in these two lines:

```
listen 80 default_server;
listen [::]:80 default_server;
```

If someone arrives at my server through the wrong domain name, I don't want them to see a page that says "Welcome to nginx", so I change the default configuration to the following, which merely responds with "Not found":

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    return 404;
}
```

4.3 Configuring nginx for django

Change `/etc/nginx/sites-available/$DOMAIN` to the following (which only differs from the one we just created in that it has the `location` block):

```
server {
    listen 80;
    listen [::]:80;
    server_name $DOMAIN www.$DOMAIN;
    root /var/www/$DOMAIN;
    location / {
        proxy_pass http://localhost:8000;
    }
}
```

Tell nginx to reload its configuration:

```
service nginx reload
```

Finally, start your Django server as we saw in the previous chapter; however, it doesn't need to listen on 0.0.0.0:8000, a mere 8000 is enough:

```
PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT \
    su $DJANGO_USER -c \
    "/opt/$DJANGO_PROJECT/venv/bin/python \
    /opt/$DJANGO_PROJECT/manage.py \
    runserver --settings=settings 8000"
```

Now go to http://$DOMAIN/ and you should see your Django project in action.

> **Warning:** We are running Django with `runserver` here, which is inappropriate for production. We are doing it only temporarily, so that you understand the concepts. We will run Django correctly in the chapter about *Gunicorn*.

Nginx receives your HTTP request. Because of the `proxy_pass` directive, it decides to just pass on this request to another server, which in our case is localhost:8000.

Now this may work for now, but we will add some more configuration which we will be necessary later. The `location` block actually becomes:

```
location / {
    proxy_pass http://localhost:8000;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_set_header X-Forwarded-For $remote_addr;
    proxy_set_header X-Forwarded-Proto $scheme;
    client_max_body_size 20m;
}
```

Here is what these configuration directives do:

proxy_set_header Host $http_host By default, the header of the request nginx makes to the backend includes `Host: localhost`. We need to pass the real `Host` to Django (i.e. the one received by nginx), otherwise Django cannot check if it's in `ALLOWED_HOSTS`.

proxy_redirect off This tells nginx that, if the backend returns an HTTP redirect, it should leave it as is. (By default, nginx assumes the backend is stupid and tries to be smart; if the backend returns an HTTP redirect that says "redirect to http://localhost:8000/somewhere", nginx replaces it with something similar to http://yourowndomain.com/somewhere". We prefer to configure Django properly instead.)

proxy_set_header X-Forwarded-For $remote_addr To Django, the request is coming from nginx, and therefore the network connection appears to be from localhost, i.e. from address 127.0.0.1 (or ::1 in IPv6). Some Django apps need to know the actual IP address of the machine that runs the web browser; they might need that for access control, or to use the GeoIP database to deliver different content to different geographical areas. So we have nginx pass the actual IP address of the visitor in the `X-Forwarded-For` header. Your Django project might not make use of this information, but it might do so in the future, and it's better to set the correct nginx configuration from now. When the time comes to use this information, you will need to configure your Django app properly; one way is to use django-ipware.

proxy_set_header X-Forwarded-Proto $scheme Another thing that Django does not know is whether the request has been made through HTTPS or plain HTTP; nginx knows that, but the request it subsequently makes to the Django backend is always plain HTTP. We tell nginx to pass this information with the `X-Forwarded-Proto` HTTP header, so that related Django functionality such as `request.is_secure()` works properly. You will also need to set `SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')` in your `settings.py`.

client_max_body_size 20m This tells nginx to accept HTTP POST requests

of up to 20 MB in length; if a request is larger nginx ignores it and returns a 413. Whether you really need that setting or not depends on whether you accept file uploads. If not, nginx's default, 1 MB, is probably enough, and it is better for protection against a denial-of-service attack that could attempt to make several large POST requests simultaneously.

This concludes the part of the chapter about nginx. If you chose nginx as your web server, you probably want to skip the next sections and go to the Chapter summary.

4.4 Installing Apache

Install Apache like this:

```
apt install apache2
```

After you install, go to your web browser and visit http://$DOMAIN/. You should see Apache's welcome page.

4.5 Configuring Apache to serve the domain

Create file `/etc/apache2/sites-available/$DOMAIN.conf` with the following contents:

```
<VirtualHost *:80>
    ServerName $DOMAIN
    ServerAlias www.$DOMAIN
    DocumentRoot /var/www/$DOMAIN
</VirtualHost>
```

Note: Again, this is not a valid Apache configuration file until you replace `$DOMAIN` with your actual domain name, such as "example.com".

Create a symbolic link in `sites-enabled`:

```
cd /etc/apache2/sites-enabled
ln -s ../sites-available/$DOMAIN.conf .
```

Hint: Symbolic links

If you don't know what symbolic links are, I have described them in *the equivalent section for nginx*.

Hint: Use a2ensite

Debian-based systems have two convenient scripts, `a2ensite`, meaning "Apache 2 enable site", and its counterpart, `a2dissite`, for disabling a site. The first one merely creates the symbolic link as above, the second one removes it. So the manual creation of the symbolic link above is purely educational, and it's usually better to save some typing by just entering this instead:

```
a2ensite $DOMAIN
```

Tell Apache to re-read its configuration:

```
service apache2 reload
```

Finally, create directory `/var/www/$DOMAIN`, and inside that directory create a file `index.html` with the following contents:

```
<p>This is the web site for $DOMAIN.</p>
```

Fire up your browser and visit http://$DOMAIN/, and you should see the page you created.

The fact that we named the Apache configuration file (in `/etc/apache2/sites-available`) `yourowndomain.com` is irrelevant; any name would have worked the same, but it's a convention to name it with the domain name. In fact, strictly speaking, we needn't even have created a separate file. The only configuration file Apache needs is `/etc/apache2/apache2.conf`. If you open that file, you will see that it contains, among others, the following line:

```
IncludeOptional sites-enabled/*.conf
```

So what it does is read all `.conf` files in that directory and process them as if their contents had been inserted in that point of `/etc/apache2/apache2.conf`.

As we noticed, if you visit http://$DOMAIN/, you see the page you created. If, however, you visit http://$SERVER_IP_ADDRESS/, you should see Apache's welcome page. If the host name (the part between "http://" and the next slash) is $DOMAIN or www.$DOMAIN, then Apache uses the configuration we specified above, because of the `ServerName` and `ServerAlias` configuration directives which contain these two names. If we use another domain name, or the server's ip address, there is no matching `VirtualHost` block in the Apache configuration, so apache uses its default configuration. That default configuration is in `/etc/apache2/sites-enabled/000-default.conf`. What makes it the default is that it is listed first; the `IncludeOptional` in `/etc/apache2/apache2.conf` reads files in alphabetical order, and `000-default.conf` has the `000` prefix to ensure it is first.

If someone arrives at my server through the wrong domain name, I don't want them to see a page that says "It works!", so I change the default configuration to the following, which merely responds with "Not found":

```
<VirtualHost *:80>
    DocumentRoot /var/www/html
    Redirect 404 /
</VirtualHost>
```

4.6 Configuring Apache for django

Change `/etc/apache2/sites-available/$DOMAIN.conf` to the following (which only differs from the one we just created in that it has the `ProxyPass` directive):

```
<VirtualHost *:80>
    ServerName $DOMAIN
    ServerAlias www.$DOMAIN
    DocumentRoot /var/www/$DOMAIN
    ProxyPass / http://localhost:8000/
</VirtualHost>
```

In order for this to work, we actually first need to enable Apache modules `proxy` and `proxy_http`, and we will take the opportunity to also enable `headers`, because we will need it soon after:

```
a2enmod proxy proxy_http headers
```

(Similarly to `a2ensite` and `a2dissite`, `a2enmod` and `a2dismod` are merely convenient ways to create and delete symbolic links that point from `/etc/apache2/mods-enabled` to `/etc/apache2/mods-available`.)

Tell Apache to reload its configuration:

```
service apache2 reload
```

Finally, start your Django server as we saw in the previous chapter; however, it doesn't need to listen on 0.0.0.0:8000, a mere 8000 is enough:

```
PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT \
    su $DJANGO_USER -c \
    "/opt/$DJANGO_PROJECT/venv/bin/python \
    /opt/$DJANGO_PROJECT/manage.py \
    runserver --settings=settings 8000"
```

Now go to http://$DOMAIN/ and you should see your Django project in action.

Warning: We are running Django with `runserver` here, which is inappropriate for production. We are doing it only temporarily, so that you understand the concepts. We will run Django correctly in the chapter about *Gunicorn*.

Apache receives your HTTP request. Because of the `ProxyPass` directive, it decides to just pass on this request to another server, which in our case is localhost:8000.

Now this may work for now, but we will add some more configuration which we will be necessary later:

```
<VirtualHost *:80>
    ServerName $DOMAIN
    ServerAlias www.$DOMAIN
    DocumentRoot /var/www/$DOMAIN
    ProxyPass / http://localhost:8000/
    ProxyPreserveHost On
    RequestHeader set X-Forwarded-Proto "http"
</VirtualHost>
```

Here is what these configuration directives do:

ProxyPreserveHost On By default, the header of the request Apache makes to the backend includes `Host: localhost` We need to pass the real `Host` to Django (i.e. the one received by Apache), otherwise Django

cannot check if it's in `ALLOWED_HOSTS`.

RequestHeader set X-Forwarded-Proto "http" Another thing that Django does not know is whether the request has been made through HTTPS or plain HTTP; Apache knows that, but the request it subsequently makes to the Django backend is always plain HTTP. We tell Apache to pass this information with the `X-Forwarded-Proto` HTTP header, so that related Django functionality such as `request.is_secure()` works properly. You will also need to set `SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')` in your `settings.py`.

This does not yet play a role because we have configured Apache to only serve plain HTTP. If we wanted it to also serve HTTPS, we would add a `<VirtualHost *:443>` block, which would contain mostly the same stuff as the `<VirtualHost *:80>` we have already defined. One of the differences is that `X-Forwarded-Proto` will be set to *"https"*.

4.7 Chapter summary

- Install your web server.
- Name the web server's configuration file with the domain name of your site.
- Put the configuration file in `sites-available` and symlink it from `sites-enabled` (don't forget to reload the web server).
- Use the `proxy_pass` (nginx) or `ProxyPass` (Apache) directive to pass the HTTP request to Django.
- Configure the web server to pass HTTP request headers `Host`, `X-Forwarded-For`, and `X-Forwarded-Proto` (Apache by default passes `X-Forwarded-For`, so there is no configuration needed for that one).
- For nginx, also configure `proxy_redirect` and `client_max_body_size`.

CHAPTER

FIVE

STATIC AND MEDIA FILES

Let's quickly make static files work. You might not understand perfectly what we're doing, but it will become very clear afterwards.

5.1 Setting up Django

First, add these statements to `/etc/opt/$DJANGO_PROJECT/settings.py`:

```
STATIC_ROOT = '/var/cache/$DJANGO_PROJECT/static/'
STATIC_URL = '/static/'
```

Remember that after each change to your settings you should, in theory, recompile:

```
/opt/$DJANGO_PROJECT/venv/bin/python -m compileall \
    /etc/opt/$DJANGO_PROJECT
```

It's not really a big deal if you forget to recompile, but we will deal with that later.

Second, create directory `/var/cache/$DJANGO_PROJECT/static/`:

```
mkdir -p /var/cache/$DJANGO_PROJECT/static
```

The `-p` parameter tells `mkdir` to create not only the directory, but, if needed, its parents as well.

Third, run `collectstatic`:

```
PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT \
    /opt/$DJANGO_PROJECT/venv/bin/python \
    /opt/$DJANGO_PROJECT/manage.py collectstatic \
    --settings=settings
```

This will copy all static files to the directory we specified in *STATIC_ROOT*. Don't worry if you don't understand it clearly, we will explain it in a minute.

5.2 Setting up nginx

Change `/etc/nginx/sites-available/$DOMAIN` to the following, which only differs from the previous version in that the new `location /static {}` block has been added at the end:

```
server {
    listen 80;
    listen [::]:80;
    server_name $DOMAIN www.$DOMAIN;
    root /var/www/$DOMAIN;
    location / {
        proxy_pass http://localhost:8000;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
        client_max_body_size 20m;
    }
```

```
    location /static/ {
        alias /var/cache/$DJANGO_PROJECT/static/;
    }
}
```

Don't forget to execute `service nginx reload` after that.

Now let's try to see if it works. **Stop the Django development server** if it is running on the server. Open your browser and visit http://$DOMAIN/. nginx should give you a 502. This is expected, since the backend is not working.

But now try to visit http://$DOMAIN/static/admin/img/icon_searchbox.png. If you have `django.contrib.admin` in `INSTALLED_APPS`, it should get a search icon (if you don't use `django.contrib.admin`, pick up another static file that you expect to see, or browse the directory `/var/cache/$DJANGO_PROJECT/static`).

Fig. 5.1 explains how this works.

The only thing that remains to clear up is what exactly these `location` blocks mean. `location /static/` means that the configuration inside the block shall apply only if the path of the URL begins with `/static/`. Likewise, `location /` applies if the path of the URL begins with a slash. However, all paths begin with a slash, so if the path begins with `/static/` both `location` blocks match the URL. Nginx only uses one `location` block. The rules with which nginx chooses the `location` block that shall apply are complicated and are described in the documentation for location, but in this particular case, nginx chooses the longest matching prefix; so if the path begins with `/static/`, nginx will choose `location /static/`.

5.3 Setting up Apache

Change `/etc/apache2/sites-available/$DOMAIN.conf` to the following:

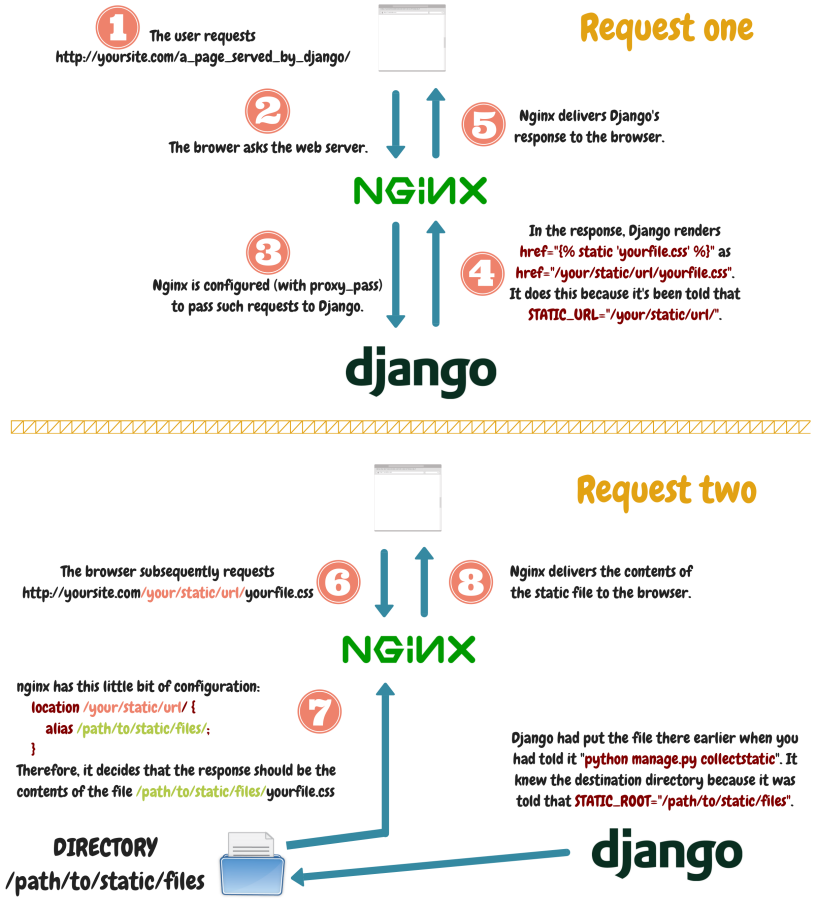


Fig. 5.1: How Django static files work in production (nginx version)

```
<VirtualHost *:80>
    ServerName $DOMAIN
    ServerAlias www.$DOMAIN
    DocumentRoot /var/www/$DOMAIN
    ProxyPass /static/ !
    ProxyPass / http://localhost:8000/
    ProxyPreserveHost On
    RequestHeader set X-Forwarded-Proto "http"
    Alias /static/ /var/cache/$DJANGO_PROJECT/static/
    <Directory /var/cache/$DJANGO_PROJECT/static/>
        Require all granted
    </Directory>
</VirtualHost>
```

Don't forget to execute `service apache2 reload` after that.

Now let's try to see if it works. **Stop the Django development server** if it is running on the server. Open your browser and visit http://$DOMAIN/. Apache should give you a 503. This is expected, since the backend is not working.

But now try to visit http://$DOMAIN/static/admin/img/icon_searchbox.png. If you have `django.contrib.admin` in `INSTALLED_APPS`, it should get a search icon (if you don't use `django.contrib.admin`, pick up another static file that you expect to see, or browse the directory `/var/cache/$DJANGO_PROJECT/static`).

Fig. 5.2 explains how this works.

Now let's examine how the configuration above produces these results. The directive `ProxyPass / http://localhost:8000/` tells Apache that, if the URL path begins with `/`, then it should pass the request to the backend. All URL paths begin with `/`, so the directive always matches. But there is also the directive `ProxyPass /static/ !`, which will match paths starting with `/static/`. When there are many matching `ProxyPass` directives, the first one wins; so for path `/static/admin/img/icon_searchbox.png`, `ProxyPass /static/ !` wins. The exclamation mark means "no proxy passing", so the

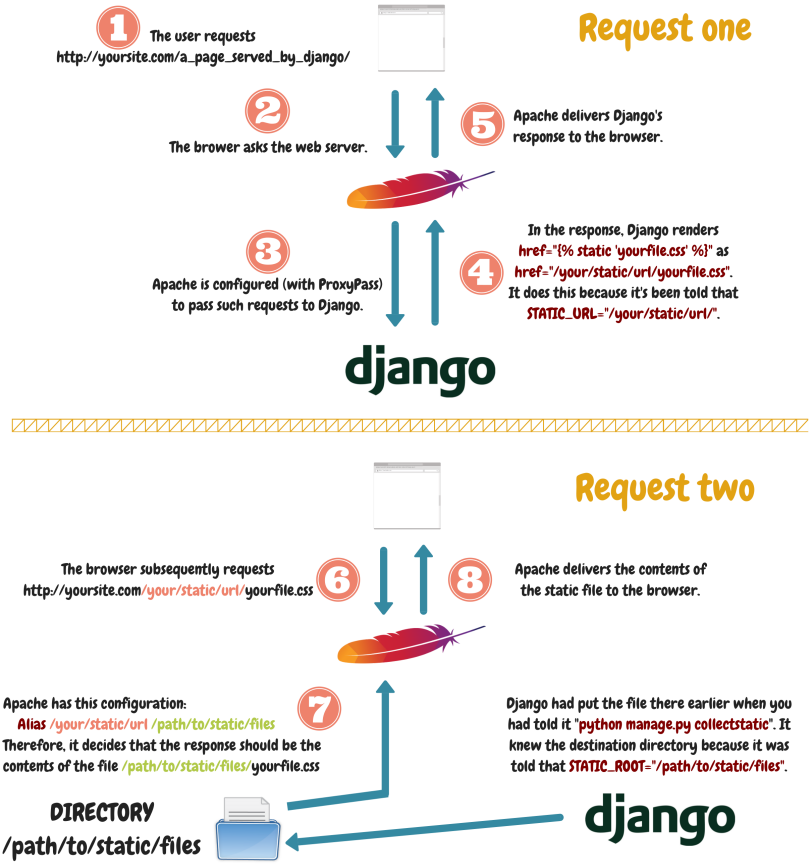


Fig. 5.2: How Django static files work in production (Apache version)

directive means "when a URL path begins with `/static/`, do not pass it to the backend". Since it is not going to be passed to the backend, Apache would normally combine it with the `DocumentRoot` and would thus try to return the file `/var/www/$DOMAIN/static/admin/img/icon_searchbox.png`, but the `Alias` directive tells it to get `/var/cache/$DJANGO_PROJECT/static/admin/img/icon_searchbox.png` instead. By default, Apache will refuse to access files in directories other than `DocumentRoot`, and will return 403, "Forbidden", in requests to access them; so we add the directive `Require all granted` for the static files directory, which means "everyone has permission to read the files".

5.4 Media files

Media files are similar to static files, so let's go through them quickly. We will store them in `/var/opt/$DJANGO_PROJECT/media`.

```
mkdir /var/opt/$DJANGO_PROJECT/media
chown $DJANGO_USER /var/opt/$DJANGO_PROJECT/media
```

One of the differences with static files is that we changed the ownership of `/var/opt/$DJANGO_PROJECT/media` to $DJANGO_USER. The reason is that Django needs to be writing there each time the user uploads a file or requests to delete a file.

Add the following to `/etc/opt/$DJANGO_PROJECT/settings.py`:

```
MEDIA_ROOT = '/var/opt/$DJANGO_PROJECT/media/'
MEDIA_URL = '/media/'
```

For nginx, add the following to `/etc/nginx/sites-available/$DOMAIN`:

```
location /media/ {
    alias /var/opt/$DJANGO_PROJECT/media/;
}
```

For Apache, add the following before `ProxyPass /`:

```
ProxyPass /media/ !
```

and the following at the end of the `VirtualHost` block:

```
Alias /media/ /var/opt/$DJANGO_PROJECT/media/
<Directory /var/opt/$DJANGO_PROJECT/media/>
    Require all granted
</Directory>
```

Recompile your settings, reload the web server, and it's ready.

5.5 File locations

Your static and media files are now served properly by the web server instead of the Django development server, and I hope you understand clearly what we've done. Let's take a break and discuss the file locations that I've chosen:

Program files	/opt/$DJANGO_PROJECT
Virtualenv	/opt/$DJANGO_PROJECT/venv
Media files	/var/opt/$DJANGO_PROJECT/media
Static files	/var/cache/$DJANGO_PROJECT/static
Configuration	/etc/opt/$DJANGO_PROJECT

There are a couple more that we haven't seen yet, but the above more or less tell the whole story.

Many people prefer a much simpler setup instead. They put everything related to their project in a single directory, which is that of their repository root, like this:

Program files	/srv/$DJANGO_PROJECT
Virtualenv	/srv/$DJANGO_PROJECT/venv
Media files	/srv/$DJANGO_PROJECT/media
Static files	/srv/$DJANGO_PROJECT/static
Configuration	/srv/$DJANGO_PROJECT/$DJANGO_PROJECT

Although this setup seems simpler, I have preferred the other one for several reasons. The first one is purely educational. When you get too used to the simple setup, you might configure always the same `STATIC_ROOT`, without really understanding what it does. The clean separation of directories should also have helped you get a grip on `PYTHONPATH` and `DJANGO_SETTINGS_MODULE`.

Separating in many directories is also cleaner and applies to many different situations. If a Django application is packaged as a `.deb` package, or as a pip-installable package, the tweak required with the split directories scheme is minimal.

Finally, separating the directories makes it easier to backup only what is needed. My backup solution (which we will see in the chapters about recovery) may exclude `/opt` and `/var/cache` from the backup. Since the static files can be regenerated, there is no need to back them up.

5.6 Chapter summary

- Set `STATIC_ROOT` to `/var/cache/$DJANGO_PROJECT/static/`.
- Set `STATIC_URL` to `/static/`.
- Set `MEDIA_ROOT` to `/var/opt/$DJANGO_PROJECT/media/`.
- Set `MEDIA_URL` to `/media/`.
- Run `collectstatic`.
- In nginx, set `location /static/ { alias /var/cache/$DJANGO_PROJECT/static/; };` likewise for media files.

- In Apache, add `ProxyPass /static/ !` before `ProxyPass /`, and add

```
Alias /static/ /var/cache/$DJANGO_PROJECT/static/
<Directory /var/cache/$DJANGO_PROJECT/static/>
    Require all granted
</Directory>
```

  Likewise for media files.

CHAPTER

SIX

GUNICORN

6.1 Why Gunicorn?

We now need to replace the Django development server with a Python application server. I will explain later why we need this. For now we need to select which Python application server to use. There are three popular servers: mod_wsgi, uWSGI, and Gunicorn.

mod_wsgi is for Apache only, and I prefer to use a method that can be used with either Apache or nginx. This will make it easier to change the web server, should such a need arise. I also find Gunicorn easier to setup and maintain.

I used uWSGI for a couple of years and was overwhelmed by its features. Many of them duplicate features that already exist in Apache or nginx or other parts of the stack, and thus they are rarely, if ever, needed. Its documentation is a bit chaotic. The developers themselves admit it: "We try to make our best to have good documentation but it is a hard work. Sorry for that." I recall hitting problems week after week and spending hours to solve them each time.

Gunicorn, on the other hand, does exactly what you want and no more. It is simple and works fine. So I recommend it unless in your particular case there

is a compelling reason to use one of the others, and so far I haven't met any such compelling reason.

6.2 Installing and running Gunicorn

We will install Gunicorn with `pip` rather than with `apt`, because the packaged Gunicorn (both in Debian 8 and Ubuntu 16.04) supports only Python 2.

```
/opt/$DJANGO_PROJECT/venv/bin/pip install gunicorn
```

Now run Django with Gunicorn:

```
su $DJANGO_USER
source /opt/$DJANGO_PROJECT/venv/bin/activate
export PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT
export DJANGO_SETTINGS_MODULE=settings
gunicorn $DJANGO_PROJECT.wsgi:application
```

You can also write it as one long command, like this:

```
PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT \
    DJANGO_SETTINGS_MODULE=settings \
    su $DJANGO_USER -c "/opt/$DJANGO_PROJECT/venv/bin/gunicorn \
    $DJANGO_PROJECT.wsgi:application"
```

Either of the two versions above will start Gunicorn, which will be listening at port 8000, like the Django development server did. Visit http://$DOMAIN/, and you should see your Django project in action.

What actually happens here is that `gunicorn`, a Python program, does something like `from $DJANGO_PROJECT.wsgi import application`. It uses `$DJANGO_PROJECT.wsgi` and `application` because we told it so in the command line. Open the file `/opt/$DJANGO_PROJECT/$DJANGO_PROJECT/wsgi.py` to see that `application` is defined there. In fact, `application` is a Python callable. Now each time Gunicorn receives an HTTP request, it calls

`application()` in a standardized way that is specified by the WSGI specification. The fact that the interface of this function is standardized is what permits you to choose between many different Python application servers such as Gunicorn, uWSGI, or mod_wsgi, and why each of these can interact with many Python application frameworks like Django or Flask.

The reason we aren't using the Django development server is that it is meant for, well, development. It has some neat features for development, such as that it serves static files, and that it automatically restarts itself whenever the project files change. It is, however, totally inadequate for production; for example, it might leave files or connections open, and it does not support processing many requests at the same time, which you really want. Gunicorn, on the other hand, does the multi-processing part correctly, leaving to Django only the things that Django can do well.

Gunicorn is actually a web server, like Apache and nginx. However, it does only one thing and does it well: it runs Python WSGI-compliant applications. It cannot serve static files and there's many other features Apache and nginx have that Gunicorn does not. This is why we put Apache or nginx in front of Gunicorn and proxy-pass requests to it. The accurate name for Gunicorn, uWSGI, and mod_wsgi would be "specialized web servers that run Python WSGI-compliant applications", but this is too long, which is why I've been using the vaguer "Python application servers" instead.

Gunicorn has many parameters that can configure its behaviour. Most of them work fine with their default values. Still, we need to modify a few. Let's run it again, but this time with a few parameters:

```
su $DJANGO_USER
source /opt/$DJANGO_PROJECT/venv/bin/activate
export PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT
export DJANGO_SETTINGS_MODULE=settings
gunicorn --workers=4 \
    --log-file=/var/log/$DJANGO_PROJECT/gunicorn.log \
    --bind=127.0.0.1:8000 --bind=[::1]:8000 \
    $DJANGO_PROJECT.wsgi:application
```

Here is what these parameters mean:

`--workers=4` Gunicorn starts a number of processes called "workers", and each process, each worker that is, serves one request at a time. To serve five concurrent requests, five workers are needed; if there are more concurrent requests than workers, they will be queued. You probably need two to five workers per processor core. Four workers are a good starting point for a single-core machine. The reason you don't want to increase this too much is that your Django project's RAM consumption is approximately proportional to the number of workers, as each worker is effectively a distinct instance of the Django project. If you are short on RAM, you might want to consider decreasing the number of workers. If you get many concurrent requests and your CPU is underused (usually meaning your Django projects do a lot of disk/database access) and you can spare the RAM, you can increase the number of workers.

> **Tip:** Check your CPU and RAM usage
>
> If your server gets busy, the Linux `top` command will show you useful information about the amount of free RAM, the RAM consumed by your Django project (and other system processes), and the CPU usage for various processes. You can read more about it in *The top command: memory management* and *The top command: CPU usage*.

`--log-file=/var/log/$DJANGO_PROJECT/gunicorn.log` I believe this is self-explanatory.

`--bind=127.0.0.1:8000` This tells Gunicorn to listen on port 8000 of the local network interface. This is the default, but we specify it here for two reasons:

1. It's such an important setting that you need to see it to know what you've done. Besides, you could be running many applications on the same server, and one could be listening on 8000, another

on 8001, and so on. So, for uniformity, always specify this.

2. We specify `--bind` twice (see below), to also listen on IPv6. The second time would override the default anyway.

`--bind=[::1]:8000` This tells Gunicorn to also listen on port 8000 of the local IPv6 network interface. This must be specified if IPv6 is enabled on the virtual server. It is not specified, things may or may not work, and the system may be a bit slower even if things work.

The reason is that the front-end web server, Apache or nginx, has been told to forward the requests to http://localhost:8000/. It will ask the the resolver what "localhost" means. If the system is IPv6-enabled, the resolver will reply with two results, `::1`, which is the IPv6 address for the localhost, and `127.0.0.1`. The web server might then decide to try the IPv6 version first. If Gunicorn has not been configured to listen to that address, then nothing will be listening at port 8000 of ::1, so the connection will be refused. The web server will then probably try the IPv4 version, which will work, but it will have made a useless attempt first.

I could make some experiments to determine exactly what happens in such cases, and not speak with "maybe" and "probably", but it doesn't matter. If your server has IPv6, you must set it up correctly and use this option. If not, you should not use this option.

6.3 Configuring systemd

The only thing that remains is to make Gunicorn start automatically. For this, we will configure it as a service in systemd.

> **Note:** Older systems don't have systemd

> systemd is relatively a novelty. It exists only in Debian 8 and later, and Ubuntu 15.04 and later. In older systems you need to start Gunicorn in another way. I recommend supervisor, which you can install with `apt install supervisor`.

The first program the kernel starts after it boots is systemd. For this reason, the process id of systemd is 1. Enter the command `ps 1` and you will probably see that the process with id 1 is `/sbin/init`, but if you look at it with `ls -lh /sbin/init`, you will see it's a symbolic link to systemd.

After systemd starts, it has many tasks, one of which is to start and manage the system services. We will tell it that Gunicorn is one of these services by creating file `/etc/systemd/system/$DJANGO_PROJECT.service`, with the following contents:

```
[Unit]
Description=$DJANGO_PROJECT

[Service]
User=$DJANGO_USER
Group=$DJANGO_GROUP
Environment="PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_
 ↪PROJECT"
Environment="DJANGO_SETTINGS_MODULE=settings"
ExecStart=/opt/$DJANGO_PROJECT/venv/bin/gunicorn \
    --workers=4 \
    --log-file=/var/log/$DJANGO_PROJECT/gunicorn.log \
    --bind=127.0.0.1:8000 --bind=[::1]:8000 \
    $DJANGO_PROJECT.wsgi:application

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

After creating that file, if you enter `service $DJANGO_PROJECT start`, it will start Gunicorn. However, it will not start it automatically at boot until we tell

it `systemctl enable $DJANGO_PROJECT`.

The `[Service]` section of the configuration file should be self-explanatory, so I will only explain the other two sections. Systemd doesn't only manage services; it also manages devices, sockets, swap space, and other stuff. All these are called units; "unit" is, so to speak, the superclass. The `[Unit]` section contains configuration that is common to all unit types. The only option we need to specify there is `Description`, which is free text. Its purpose is only to show in the UI of management tools. Although $DJANGO_PROJECT will work as a description, it's better to use something more verbose. As the systemd documentation says,

> "Apache2 Web Server" is a good example. Bad examples are "high-performance light-weight HTTP server" (too generic) or "Apache2" (too specific and meaningless for people who do not know Apache).

The `[Install]` section tells systemd what to do when the service is enabled. The `WantedBy` option specifies dependencies. If, for example, we wanted to start Gunicorn before nginx, we would specify `WantedBy=nginx.service`. This is too strict a dependency, so we just specify `WantedBy=multi-user.target`. A target is a unit type that represents a state of the system. The multi-user target is a state all GNU/Linux systems reach in normal operations. Desktop systems go beyond that to the "graphical" target, which "wants" a multi-user system and adds a graphical login screen to it; but we want Gunicorn to start regardless whether we have a graphical login screen (we probably don't, as it is a waste of resources on a server).

As I already said, you tell systemd to automatically start the service at boot (and automatically stop it at system shutdown) in this way:

```
systemctl enable $DJANGO_PROJECT
```

Do you remember that in nginx and Apache you enable a site just by creating a symbolic link to `sites-available` from `sites-enabled`? Likewise, `systemctl enable` does nothing but create a symbolic link. The dependencies we have specified in the `[Install]` section of the configuration file de-

termine where the symbolic link will be created (sometimes more than one symbolic links are created). After you enable the service, try to restart the server, and check that your Django project has started automatically.

As you may have guessed, you can disable the service like this:

```
systemctl disable $DJANGO_PROJECT
```

This does not make use of the information in the `[Install]` section; it just removes all symbolic links.

6.4 More about systemd

While I don't want to bother you with history, if you don't read this section you will eventually get confused by the many ways you can manage a service. For example, if you want to tell nginx to reload its configuration, you can do it with either of these commands:

```
systemctl reload nginx
service nginx reload
/etc/init.d/nginx reload
```

Before systemd, the first program that was started by the kernel was `init`. This was much less smart than systemd and did not know what a "service" is. All `init` could do was execute programs or scripts. So if we wanted to start a service we would write a script that started the service and put it in `/etc/init.d`, and enable it by linking it from `/etc/rc2.d`. When `init` brought the system to "runlevel 2", the equivalent of systemd's multi-user target, it would execute the scripts in `/etc/rc2.d`. Actually it wasn't `init` itself that did that, but other programs that `init` was configured to run, but this doesn't matter. What matters is that the way you would start, stop, or restart nginx, or tell it to reload its configuration, or check its running status, was this:

```
/etc/init.d/nginx start
/etc/init.d/nginx stop
/etc/init.d/nginx restart
/etc/init.d/nginx reload
/etc/init.d/nginx status
```

The problem with these commands was that they might not always work correctly, mostly because of environment variables that might have been set, so the `service` script was introduced around 2005, which, as its documentation says, runs an init script "in as predictable an environment as possible, removing most environment variables and with the current working directory set to /." So a better alternative for the above commands was

```
service nginx start
service nginx stop
service nginx restart
service nginx reload
service nginx status
```

The new way of doing these with systemd is the following:

```
systemctl start nginx
systemctl stop nginx
systemctl restart nginx
systemctl reload nginx
systemctl status nginx
```

Both `systemctl` and `service` will work the same with your Gunicorn service, because `service` is a backwards compatible way to run `systemctl`. You can't manage your service with an `/etc/init.d` script, because we haven't created any such script (and it would have been very tedious to do so, which is why we preferred to use supervisor before we had systemd). For nginx and Apache, all three ways are available, because most services packaged with the operating system are still managed with init scripts, and systemd has a backwards compatible way of dealing with such scripts. In future versions of Debian and Ubuntu, it is likely that the init scripts will be replaced with systemd configu-

ration files like the one we wrote for Gunicorn, so the `/etc/init.d` way will cease to exist.

Of the remaining two newer ways, I don't know which is better. `service` has the benefit that it exists in non-Linux Unix systems, such as FreeBSD, so if you use both GNU/Linux and FreeBSD you can use the same command in both. The `systemctl` version may be more consistent with other systemd commands, like the ones for enabling and disabling services. Use whichever you like.

6.5 The top command: memory management

If your server gets busy and you wonder whether its RAM and CPU are enough, the Linux `top` command is a useful tool. Execute it simply by entering `top`. You can exit `top` by pressing `q` on the keyboard.

When you execute `top` you will see an image similar to Fig. 6.1.

Let's examine **available RAM** first, which in Fig. 6.1 is indicated in the red box. The output of `top` is designed so that it fits in an 80-character wide terminal. For the RAM, the five values (total, used, free, buffers, and cached) can't fit on the line that is labeled "KiB Mem", so the last one has been moved to the line below, that is, the "cached Mem" indication belongs in "KiB Mem" and not in "KiB Swap".

The "total" amount of RAM is simply the total amount of RAM; it is as much as you asked your virtual server to have. The "used" plus the "free" equals the total. Linux does heavy caching, which I explain below, so the "used" should be close to the total, and the "free" should be close to zero.

Since RAM is much faster than the disk, Linux caches information from the disk in RAM. It does so in a variety of ways:

- If you open a file, read it, close it, then you open it again and read it again, the second time it will be much faster; this is because Linux has cached the contents of the file in RAM.

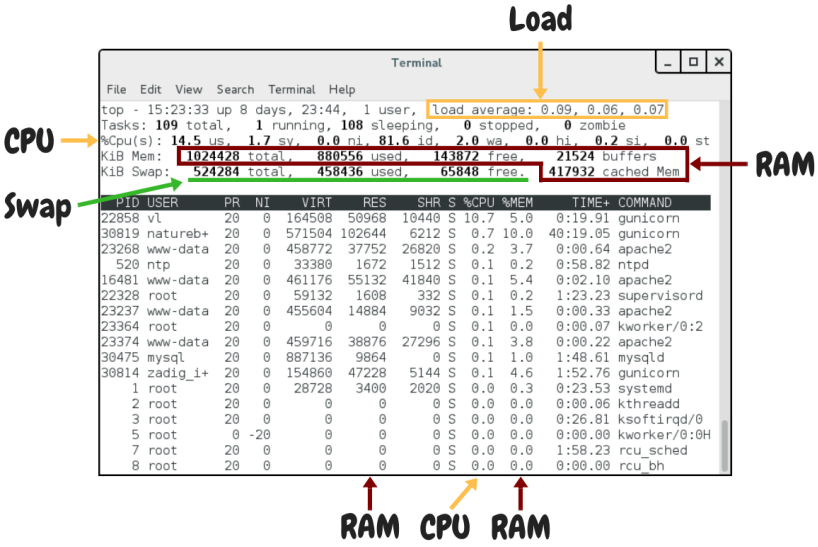


Fig. 6.1: The `top` command

- Whenever you write a file, you are likely to read it again, so Linux caches it.
- In order to speed up disk writing, Linux doesn't actually write to the disk when your program says `f.write(data)`, not even when you close the file, not even when your program ends. It keeps the data in the cache and writes it later, attempting to optimize disk head movement. This is why some data may be lost when the system is powered off instead of properly shut down.

The part of RAM that is used for Linux's disk cache is what `top` shows as "buffers" and "cached". Buffers is also a kind of cache, so it is the sum of "buffers" and "cache" that matters (the difference between "buffers" and "cached" doesn't really matter unless you are a kernel developer). "Buffers" is usually negligible, so it's enough to only look at "cache".

Linux doesn't want your RAM sitting down doing nothing, so if there is RAM available, it will use it for caching. Give it more RAM and it will cache more. If your server has a substantial amount of RAM labeled "free", it may mean that you have so much RAM that Linux can't fill it in even with its disk cache. This probably means the machine is larger than it needs to be, so it's a waste of resources. If, on the other hand, the cache is very small, this may mean that the system is short on RAM. On a healthy system, the cache should be 20–50% of RAM.

Since we are talking about RAM, let's also examine the **amount of RAM used by processes**. By default `top` sorts processes by CPU usage, but you can type `M` (Shift + `m`) to sort by memory usage (you can go back to sort by CPU usage by typing `P`). The RAM used by each process is indicated by the "RES" column in KiB and the "%MEM" column in percentage.

There are two related columns; "VIRT", for virtual memory, and "SHR", for shared memory. First of all, you need to forget the Microsoft terminology. Windows calls "virtual memory" what everyone else calls "swap space"; and what everyone else calls "virtual memory" is a very different thing from swap space. In order to better understand what virtual memory is, let's see it with this C program (it doesn't matter if you don't speak C):

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

int main() {
    char c;
    void *p;

    /* Allocate 2 GB of memory */
    p = malloc(2L * 1024 * 1024 * 1024);
    if (!p) {
        fprintf(stderr, "Can't allocate memory: %s\n",
                strerror(errno));
        exit(1);
```

```
    }

    /* Do nothing until the user presses Enter */
    fputs("Press Enter to continue...", stderr);
    while((c = fgetc(stdin)) != EOF && c != '\n')
        ;

    /* Free memory and exit */
    free(p);
    exit(0);
}
```

When I run this program on my laptop, and while it is waiting for me to press Enter, this is what `top` shows about it:

```
. PID ...     VIRT    RES    SHR S  %CPU %MEM ... COMMAND
13687 ... 2101236    688    612 S   0.0  0.0 ... virtdemo
```

It indicates 2 GB VIRT, but actually uses less than 1 MB of RAM, while swap usage is still at zero. Overall, running the program has had a negligible effect on the system. The reason is that the `malloc` function has only allocated virtual memory; "virtual" as in "not real". The operating system has provided 2 GB of virtual address space to the program, but the program has not used any of that. If the program had used some of this virtual memory (i.e. if it had written to it), the operating system would have automatically allocated some RAM and would have mapped the used virtual address space to the real address space in the RAM.

So virtual memory is neither swap nor swap plus RAM; it's virtual. The operating system maps only the used part of the process's virtual memory space to something real; usually RAM, sometimes swap. Many programs allocate much more virtual memory than they actually use. For this reason, the VIRT column of `top` is not really useful. The RES column, that stands for "resident", indicates the part of RAM actually used.

The SHR column indicates how much memory the program potentially shares

with other processes. Usually all of that memory is included in the RES column. For example, in Fig. 6.1, there are four `apache2` processes which I show again here:

```
. PID ...    VIRT    RES    SHR S  %CPU %MEM ... COMMAND
23268 ...  458772  37752  26820 S   0.2  3.7 ... apache2
16481 ...  461176  55132  41840 S   0.1  5.4 ... apache2
23237 ...  455604  14884   9032 S   0.1  1.5 ... apache2
23374 ...  459716  38876  27296 S   0.1  3.8 ... apache2
```

It is unlikely that the total amount of RAM used by these four processes is the sum of the RES column (about 140 MB); it is more likely that something like 9 MB is shared among all of them, which would bring the total to about 110 MB. Maybe even less. They might also be sharing something (such as system libraries) with non-apache processes. It is not really possible to know how much of the memory marked as shared is actually being shared, and by how many processes, but it is something you need to take into account in order to explain why the total memory usage on your system is less than the sum of the resident memory for all processes.

Let's now talk about **swap**. Swap is disk space used for temporarily writing (swapping) RAM. Linux uses it in two cases. The first one is if a program has actually used some RAM but has left it unused for a long time. If a process has written something to RAM but has not read it back for several hours, it means the RAM is being wasted. Linux doesn't like that, so it may save that part of RAM to the disk (to the swap space), which will free up the RAM for something more useful (such as caching). This is the case in Fig. 6.1. The system is far from low on memory, and yet it has used a considerable amount of swap space. The only explanation is that some processes have had unused data in RAM for too long. When one of these processes eventually attempts to use swapped memory, the operating system will move it from the swap space back to the RAM (if there's not enough free RAM, it will swap something else or discard some of its cache).

The second case in which Linux will use swap is if it's low on memory. This is a bad thing to happen and will greatly slow down the system, sometimes

to a grinding halt. You can understand that this is the case from the fact that swap usage will be considerable while at the same time the free and cached RAM will be very low. Sometimes you will be unable to even run `top` when this happens.

Whereas in Windows the swap space (confusingly called "virtual memory") is a file, on Linux it is usually a disk partition. You can find out where swap is stored on your system by examining the contents of file `/proc/swaps`, for example by executing `cat /proc/swaps`. (The "files" inside the `/proc` directory aren't real; they are created by the kernel and they do not exist on the disk. `cat` prints the contents of files, similar to `less`, but does not paginate.)

6.6 The top command: CPU usage

The third line of `top` has eight numbers which add up to 100%. They are user, system, nice, idle, waiting, hardware interrupts, software interrupts, and steal, and indicate where the CPU spent its time in the last three seconds:

- **us** (user) and **sy** (system) indicate how much of its time the processor was running programs in user mode and in kernel mode. Most code runs in user mode; but when a process asks the Linux kernel to do something (allocate memory, access the disk, network, or other device, start another process, etc.), the kernel switches to kernel mode, which means it has some priviliges that user mode doesn't have. (For example, kernel mode has access to all RAM and can modify the mapping between the processes' virtual memory and RAM/swap; whereas user mode simply has access to the virtual address space and doesn't know what happens behind the scenes.)
- **ni** (nice) indicates how much of its time the processor was running with a positive "niceness" value. If many processes need the CPU at the same time, a "nice" process has lower priority. The "niceness" is a number up to 19. A process with a "niceness" of 19 will practically only run when the CPU would otherwise be idle. For example,

the GNOME desktop environment's Desktop Search finds stuff in your files, and it does so very fast because it uses indexes. These indexes are updated in the background by the "tracker" process, which runs with a "niceness" of 19 in order to not make the rest of the system slower. Processes may also run with a negative niceness (up to -20), which means they have higher priority. In the list of processes, the NI column indicates the "niceness". Most processes have the default zero niceness, and it is unlikely you will ever need to know more about all that.

- **id** (idle) and **wa** (waiting) indicate how much time the CPU was sitting down doing nothing. "Waiting" is a special case of idle; it means that while the CPU was idle there was at least one process waiting for disk I/O. A high value of "waiting" indicates heavy disk usage.
- The meaning of time spent in **hi** (hardware interrupts) and **si** (software interrupts) is very technical. If this is non-negligible, it indicates heavy I/O (such as disk or network).
- **st** (steal) is for virtual machines. When nonzero, it indicates that for that amount of time the virtual machine needed to run something on the (virtual) CPU, but it had to wait because the real CPU was unavailable, either because it was doing something else (e.g. servicing another virtual machine on the same host) or because of reaching the CPU usage quota.

If the machine has more than one CPUs or cores, the "%Cpu(s)" line of `top` shows data collectively for all CPUs; but you can press `1` to toggle between that and showing information for each individual CPU.

In the processes list, the %CPU column indicates the amount of time the CPU was working for that process, either in user mode or in kernel mode (when kernel code is running, most of the time it is in order to service a process, so this time is accounted for in the process). The %CPU column can add up to more than 100% if you have more than one cores; for four cores it can add up to 400% and so on.

Finally, let's discuss about the CPU load. When your system is doing nothing,

the CPU load is zero. If there is one process using the CPU, the load is one. If there is one process using the CPU and another process that wants to run and is queued for the CPU to become available, the load is two. The three numbers in the orange box in Fig. 6.1 are the load average in the last one, five, and 15 minutes. The load average should generally be less than the number of CPU cores, and preferably under 0.7 times the number of cores. It's OK if it spikes sometimes, so the load average for the last minute can occasionally go over the number of cores, but the 5- or 15-minute average should stay low. For more information about the load average, there's an excellent blog post by Andre Lewis, Understanding Linux CPU Load - when should you be worried?

6.7 Chapter summary

- Install `gunicorn` in your virtualenv.
- Create file `/etc/systemd/system/$DJANGO_PROJECT.service` with these contents:

```
[Unit]
Description=$DJANGO_PROJECT

[Service]
User=$DJANGO_USER
Group=$DJANGO_GROUP
Environment="PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/
 ↪$DJANGO_PROJECT"
Environment="DJANGO_SETTINGS_MODULE=settings"
ExecStart=/opt/$DJANGO_PROJECT/venv/bin/gunicorn \
    --workers=4 \
    --log-file=/var/log/$DJANGO_PROJECT/gunicorn.log \
    --bind=127.0.0.1:8000 --bind=[::1]:8000 \
    $DJANGO_PROJECT.wsgi:application
```

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

- Enable the service with `systemctl enable $DJANGO_PROJECT`, and start/stop/restart it or get its status with `systemctl $COMMAND $DJANGO_PROJECT`, where $COMMAND is start, stop, restart or status.

CHAPTER

SEVEN

PRODUCTION SETTINGS

So far the only thing we've done in our production settings was to setup `ALLOWED_HOSTS`. We still have some work to do. It is absolutely essential to setup email and the secret key, it is a good idea to setup logging, and we may also need to setup caching. Most installations will not need anything beyond these.

7.1 Email

Even if your Django application does not use email at all, you must still set it up. The reason is that your code has bugs. Even if it does not have bugs, your server will eventually run into an error condition, such as no disk space, out of memory, or something else going wrong. In many of these cases, Django will throw a "500 error" to the user and will try to email you. You really need to receive that email.

First, you need a mail server to which you can connect and ask to send an email. Such a mail server is called a "smarthost". The mechanism with which Django connects to the smarthost is pretty much the same as the one with which your desktop or mobile mail client connects to an outgoing mail server. However, the term "outgoing mail server" is mostly used for mailing

software, and "smarthost" is used when some unattended software like your Django app sends email. You can often, but not always, use your outgoing mail server as smarthost.

I'm using Runbox for my email, and I also use it as a smarthost. There are many other providers, one of the most popular being Gmail (I believe, however, that it's not possible to use Gmail as a smarthost if all you have is a free account, and even if it is possible, it is hard to setup).

Let's set it up and then we will discuss more. Add the following to `/etc/opt/$DJANGO_PROJECT/settings.py`:

```
SERVER_EMAIL = 'noreply@$DOMAIN'
DEFAULT_FROM_EMAIL = 'noreply@$DOMAIN'
ADMINS = [
    ('$ADMIN_NAME', '$ADMIN_EMAIL_ADDRESS'),
]
MANAGERS = ADMINS

EMAIL_HOST = '$EMAIL_HOST'
EMAIL_HOST_USER = '$EMAIL_HOST_USER'
EMAIL_HOST_PASSWORD = '$EMAIL_HOST_PASSWORD'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

SERVER_EMAIL is the email address from which emails with error messages appear to come from. It is set in the "From:" field of the email. The default is "root@localhost", and while "root" is OK, "localhost" is not, and some mail servers may refuse the email. The domain name where your Django application runs is usually OK, but if this doesn't work you can use any other valid domain. The domain of your email address should work properly.

If your Django project does not send any emails (other than the error messages Django will send anyway), DEFAULT_FROM_EMAIL does not need to be specified. If it does send emails, it may be using django.core.mail.EmailMessage. In order to specify what will be in the

"From:" field of the email, `EmailMessage` accepts a `from_email` argument at initialization; if this is unspecified, it will use `DEFAULT_FROM_EMAIL`. So `DEFAULT_FROM_EMAIL` is exactly what it says: the default `from_email` of `EmailMessage`. It's a good idea to specify this, because even if your Django project does not send emails today, it may well do so tomorrow, and the default, "webmaster@localhost", is not a good option. Remember that with `EmailMessage` you are likely to send email to your users, and it should be something nice. "noreply@$DOMAIN" is usually fine.

ADMINS is a list of people to whom error messages will be sent. Make sure your name and email address are listed there, and also add any fellow administrators. MANAGERS is similar to `ADMINS`, but for broken link notifications, and usually you just need to set it to the same values as `ADMINS`.

The settings starting with `EMAIL_` describe how Django will connect and authenticate to the mail server. Django will connect to EMAIL_HOST and authenticate using EMAIL_HOST_USER and EMAIL_HOST_PASSWORD. Needless to say, I have used placeholders that start with a dollar sign, and you need to replace these with actual values. Mine are usually these:

```
EMAIL_HOST = 'mail.runbox.com'
EMAIL_HOST_USER = 'smarthostclient%antonischristofides.com'
EMAIL_HOST_PASSWORD = 'topsecret'
```

However, the details depend on the provider and the account type you have. I don't use my personal email, which is antonis@antonischristofides.com (Runbox requires you to change @ to % when you use it as a user name for login), because my personal password would then be in many `settings.py` files in many deployed Django projects, and I'm not the only administrator of these servers (and even if I were, I wouldn't know when I would invite another one). So I created another user (subaccount in Runbox parlance), "smarthostclient", which I use for that purpose.

There are three ports used for sending email: 25, 465, and 587. The sender (Django in our case, or your mail client when you send email) connects to a mail server and gives the email to it; the mail server then delivers the email

to another mail server, and so on, until the destination is reached. In the old times both the initial submission and the communication between mail servers was through port 25. Nowadays 25 is mostly used for communication between mail servers only. If you try to use port 25 (which is the default setting for EMAIL_PORT), it's possible that the request will get stuck in firewalls, and even if does reach the mail server, the mail server is likely to refuse to send the email. This is because spam depends much on port 25, so policies about this port are very tight.

The other two ports for email submission are 465 and 587. 465 uses encryption; just as 80 is for unencrypted HTTP and 443 is for encrypted HTTP, 25 is for unencrypted SMTP and 465 is for encrypted SMTP. However, 465 is deprecated in favour of 587, which can handle both unencrypted and encrypted connections. The client (Django in our case) connects to the server at port 587, they start talking unencrypted, and the client may tell the server "I want to continue with encryption", and then they continue with encryption. Obviously this is done before authentication, which requires the password to be transmitted.

There are thus two methods to start encryption; one is implicit and the other one is explicit. When you connect to port 465, which always works encrypted, the encryption starts implicitly. When you connect to port 587, the two peers (the client and the server) start talking unencrypted, and at some point the client explicitly tells the server "I want to continue with encryption". Computer people often use "SSL" for implicit encryption and "TLS" for explicit, however this is inaccurate; SSL and TLS are encryption protocols, and do not refer to the method used to initiate them; you could have implicit TLS or explicit SSL. Django uses this inaccurate parlance in its settings, where EMAIL_USE_TLS and EMAIL_USE_SSL are used to specify whether, respectively, the connection will use explicit or implicit encryption. `EMAIL_USE_TLS = True` should be used with `EMAIL_PORT = 587`, and `EMAIL_USE_SSL = True` with `EMAIL_PORT = 465`.

To test your settings, start a shell from your Django project:

```
PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT \
DJANGO_SETTINGS_MODULE=settings \
su $DJANGO_USER -c \
"/opt/$DJANGO_PROJECT/venv/bin/python \
/opt/$DJANGO_PROJECT/manage.py shell"
```

and enter these commands:

```
from django.conf import settings
from django.core.mail import send_mail

admin_emails = [x[1] for x in settings.ADMINS]
send_mail("Test1557", "Hello", settings.SERVER_EMAIL,
          admin_emails)
```

If something goes wrong, `send_mail` will raise an exception; otherwise you should receive the email.

Because of spam, mail servers are often very picky about which emails they will accept. It's possible that even if your smarthost accepts the email, the next mail server may refuse it. For example, I made some experiments using `from_email='noreply@example.com'`, `EMAIL_HOST = 'mail.runbox.com'`, and recipient anthony@itia.ntua.gr (an old email address of mine). In that case, Runbox accepted the email and subsequently attempted to deliver it to the mail server of ntua.gr, which rejected it because it didn't like the sender (noreply@example.com; I literally used "example.com", and ntua.gr didn't like that domain). When something like this happens, the test we made above with `send_mail` will appear to work, because `send_mail` manages to deliver the email to the smarthost, and the error occurs after that; not only will we never receive the email, but it is also likely that we will not receive the failure notification (the returned email), so it's often hard to know what went wrong and we need to guess.

One thing you can do to lessen the probability of error is to make sure that the recipient (or at least one of the recipients) has an email address served by the provider who provides the smarthost. In my case, the smarthost

is `mail.runbox.com`, and the recipient is antonis@antonischristofides.com, and the email for domain antonischristofides.com is served by Runbox. It is unlikely that `mail.runbox.com` would accept an email addressed to antonis@antonischristofides.com if another Runbox server were to subsequently refuse it. If something like this happened, I believe it would be a configuration error on behalf of Runbox. But it's very normal that `mail.runbox.com` will accept an email which will subsequently be refused by ntua.gr or Gmail or another provider downstream.

7.2 Debug

After you have configured email and verified it works, you can now turn off DEBUG:

```
DEBUG = False
```

Now it's good time to verify that error emails do indeed get sent properly. You can do so by deliberately causing an internal server error. A favourite way of mine is to temporarily rename a template file and make a related request, which will raise a `TemplateDoesNotExist` exception. Your browser should show the "server error" page. Don't forget to rename the template file back to what it was. By the time you finish doing that, you should have received the email with the full trace.

7.3 Using a local mail server

Usually I don't configure Django to deliver to the smarthost; instead, I install a mail server locally, have Django deliver to the local mail server, and configure the local mail server to send the emails to the smarthost. There are several reasons why installing a local mail server is better:

1. Your server, like all Unix systems, has a scheduler, `cron`, which is configured to run certain programs at certain times. For example, directory `/etc/cron.daily` contains scripts that are executed once per day. Whenever a program run by `cron` throws an error message, `cron` emails that error message to the administrator. `cron` always works with a local mail server. If you don't install a local mail server, you will miss these error messages. We will later use `cron` to clear sessions and to backup the server, and we don't want to miss any error messages.

2. While Django attempts to send an error email, if something goes wrong, it fails silently. This behaviour is appropriate (the system is in error, it attempts to email its administrator with the exception, but sending the email also results in an error; can't do much more). Suppose, however, that when you try to verify, as we did in the previous section, that error emails work, you find out they don't work. What has gone wrong? Nothing is written in any log. Intercepting the communication with `ngrep` won't work either, because it's usually encrypted. If you use a locally installed mail server, you will at least be able to look at the local mail server's logs.

3. Sending an error email might take long. The communication line might be slow, or a firewall or the DNS could be misbehaving, and it might take several seconds, or even a minute, before Django manages to establish a connection to the remote mail server. During this time, the browser will be in a waiting state, and a Gunicorn process will be occupied. Some people will recommend to send emails from celery workers, but this is not possible for error emails. In addition, there is no reason to install and program celery just for this reason. If we use a local mail server, Django will deliver the email to it very fast and finish its job, and the local mail server will queue it and send it when possible.

While the most popular mail servers for Debian and Ubuntu are exim and postfix, I don't recommend them. Mail servers are strange beasts. They have large and tricky configuration files, because they can do a hell of things. You will have a hard time understanding the necessary configuration (which is buried under a hell of other configuration), and if something goes wrong you

will have a hard time debugging it. I also see no great educational value in learning it. I used to run mail servers for years but I've got ridden of all of them; it's not worth the effort when I can do the same thing at Runbox for € 30 per year.

Instead, we are going to use `dma` (nothing to do with direct memory access; this is the DragonFly Mail Agent). It's a small mail server that only does what we want; it collects messages in a queue, and sends them to a smarthost. It is much easier to configure than the real thing. Install it like this:

```
apt install dma
```

It will ask you a couple of questions:

System mail name You should probably use $DOMAIN here. If that doesn't work, you can try to use the domain of your email address.

Smarthost This is the remote mail server, the smarthost, that is; the one we had specified in Django's `EMAIL_HOST`.

Next, open `/etc/dma/dma.conf` in an editor, and uncomment or edit these directives:

```
PORT 587
AUTHPATH /etc/dma/auth.conf
SECURETRANSFER
STARTTLS
```

(If your smarthost uses implicit encryption, you need to specify `PORT 465` instead, and omit the `STARTTLS`.)

Next, open `/etc/dma/auth.conf` and add this line:

```
$EMAIL_USER|$EMAIL_HOST:$EMAIL_PASSWORD
```

(These are placeholders of course, which you need to replace.)

Next, open `/etc/aliases` and add this line:

```
root: $ADMIN_EMAIL_ADDRESS
```

Finally, open `/etc/mailname` in an editor and make sure it contains a single line which contains your domain ($DOMAIN).

Let's test it to see if it works:

```
sendmail $ADMIN_EMAIL_ADDRESS
```

This will pause for input. Type a short email message, and end it with a line that contains a single fullstop. Check `/var/log/mail.log` to verify it has been delivered to the smarthost (if it says "delivery successful" it's OK, even if it's preceded by a warning message about the authentication mechanism), and verify that you have received it.

The next step is to configure Django. You might think that we would set `EMAIL_HOST = 'localhost'` and `EMAIL_PORT = 25`, but this is not what we will do. `dma` does not listen on port 25 or on any other port. The only way to send emails with it is by using the `sendmail` command. Traditionally this has been the easiest and most widely available way to send emails in Unix, and it is also what `cron` uses. (In the old times, when `sendmail` was the only existing mail server, the practice of using the `sendmail` command was standardized, so today all mail servers create a `sendmail` command when they are installed, which is usually a symbolic link to something else). We will install a Django email backend that sends emails in the same way.

```
/opt/$DJANGO_PROJECT/venv/bin/pip install django-sendmail-backend
```

The only Django configuration we need is this:

```
EMAIL_BACKEND = 'django_sendmail_backend.backends.EmailBackend'
```

The `dma` configuration should have been obvious, except for `/etc/aliases` and `/etc/mailname`. These are not dma-specific, they are also used by exim, postfix, and most other mail servers, and `/etc/mailname` may also be used by other programs.

`/etc/aliases` specifies aliases for email addresses. If `cron` decides it needs to send an email, the recipient will most likely be a mere `root`. The line we added specifies that `root` should be translated to your actual email address. For Django, `/etc/aliases` doesn't matter, since Django will get the recipient email address from the `ADMINS` and `MANAGERS` settings.

If a program somehow needs to know the domain used for the email of the system, it usually takes it from `/etc/mailname`. Setting that to `$DOMAIN` should be fine, but if this doesn't work, you can try setting it to the domain of your email address.

7.4 Secret key

Django uses the SECRET_KEY in several cases, for example, when digitally signing sessions in cookies. If it leaks, attackers might be able to compromise your system. You should not use the `SECRET_KEY` you use in development, because that one is easy to leak, and because many developers often have access to it, whereas they should not have access to the production `SECRET_KEY`.

You can create a secret key in this way:

```
import sys

from django.utils.crypto import get_random_string

sys.stdout.write(get_random_string(50))
```

7.5 Logging

Even if your Django apps do no logging, they eventually will. At some point one of your users is going to cause an error which you will be unable to reproduce in the development environment, so you will introduce

some logging calls. It makes sense to configure logging so that it is ready for that time. You need a configuration that will write log messages in `/var/log/$DJANGO_PROJECT/$DJANGO_PROJECT.log`, and here it is:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '[%(asctime)s] %(levelname)s: '
                      '%(message)s',
        }
    },
    'handlers': {
        'file': {
            'class': 'logging.handlers.'
                     'TimedRotatingFileHandler',
            'filename': '/var/log/$DJANGO_PROJECT/'
                        '$DJANGO_PROJECT.log',
            'when': 'midnight',
            'backupCount': 60,
            'formatter': 'default',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'INFO',
    },
}
```

Here is the meaning of the various items:

version This is reserved for the future; for now, it should always be 1.

disable_existing_loggers Django already has a default logging configuration. If `disable_existing_loggers` is `True` (the default), then this configuration will override Django's default, otherwise it will work in addition to the default. We really want Django's default configuration,

which is to email critical errors to the administrators.

root This defines the root logger. You can specify very complicated logging schemes, where different loggers will be logging using different handlers and different formatters. However, as long as our system is small, we only need to specify a single logger, the root logger, which uses a single handler (the "file" handler) with a single formatter (the "default" formatter). In this example I have specified `'level': 'INFO'`, which means the logger will ignore messages with a lower priority (the only lower priority is `DEBUG`, and the higher priorities are `WARNING`, `ERROR` and `CRITICAL`). You can change this as needed, however `INFO` is reasonable to begin with.

handlers Here we define the "file" handler, whose class is `logging.TimedRotatingFileHandler`. This essentially logs to a file, but it has the added benefit that each midnight it starts a new log file, renames the old one, and deletes log files older than 60 days. In this way it is very unlikely that your disk will fill up because of the growing log files escaping your attention.

formatters This defines a formatter named "default". In a system where I'm using this logging configuration, I have this code:

```
import logging

# ...

logging.info('Notifying user {} about the agrifields of '
             'user {}'.format(user, owner))
```

and it produces this line in the log file:

```
[2016-11-29 04:40:02,880] INFO: Notifying user aptiko␣
 ↪about the agrifields of user aptiko
```

7.6 Caching

The only other setting I expect you to set to a different value from development is CACHES. How you will set it depends on your needs. I usually want my caches to persist across reboots, so I specify this:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.'
                   'FileBasedCache',
        'LOCATION': '/var/cache/$DJANGO_PROJECT/cache',
    }
}
```

You also need to create the directory and give it the necessary permissions:

```
mkdir /var/cache/$DJANGO_PROJECT/cache
chown $DJANGO_USER /var/cache/$DJANGO_PROJECT/cache
```

7.7 Recompile your settings

Remember that Django runs as $DJANGO_USER and does not (and should not) have permission to write in directory /etc/opt/$DJANGO_PROJECT, which is owned by root. Therefore it can't write the Python 2 compiled file settings.pyc, or the Python 3 compiled files directory __pycache__. In theory you should be compiling it each time you make a change to your settings:

```
/opt/$DJANGO_PROJECT/venv/bin/python -m compileall \
    /etc/opt/$DJANGO_PROJECT
```

Of course it's not possible to remember to do this every single time you change something in the settings. There are two solutions to this. The first solution, which is fine, is to ignore the problem. If the compiled file is absent

or outdated, Python will compile the source file on the spot. This will happen whenever each gunicorn worker starts, which is only when you start or restart gunicorn, and it costs less than 1 ms. It's really negligible.

The second solution is to create a script `/usr/local/sbin/restart-$DJANGO_PROJECT`, with the following contents:

```
#!/bin/bash
set -e
/opt/$DJANGO_PROJECT/venv/bin/python -m compileall -q \
     -x /opt/$DJANGO_PROJECT/venv/ /opt/$DJANGO_PROJECT \
     /etc/opt/$DJANGO_PROJECT
service $DJANGO_PROJECT restart
```

You must make that script executable:

```
chmod 755 /usr/local/sbin/restart-$DJANGO_PROJECT
```

You might object that we don't want users other than root to be able to recompile the Python files or to restart the gunicorn service. The answer is that they won't be able. They will be able to execute the script, but when the script arrives at the point where it compiles the Python files, they will be denied permission to write the compiled Python files to the directory; and if the script ever arrives at the last line, again systemd will deny to restart the service. Making a script non-executable doesn't achieve anything security-wise; a malicious user could simply copy it and make the copy executable.

From now on, whenever you want to restart gunicorn, instead of `service $DJANGO_PROJECT restart`, you can be using `restart-$DJANGO_PROJECT`, which will run the above script. The `set -e` command tells bash to stop executing the script when an error occurs, and the `-q` parameter to `compileall` tells to not print the list of files compiled.

7.8 Clearing sessions

If you use `django.contrib.sessions`, Django stores session data in the database (unless you use using a different SESSION_ENGINE). Django does not automatically clean up the sessions table, so most of the sessions remain in the database even after they expire. I've seen sessions tables in small deployments of only a few requests per minute grow to several hundreds of GB through the years. You can manually remove expired sessions by executing `python manage.py clearsessions`.

To make sure your sessions are being cleared regularly, create file `/etc/cron.daily/$DJANGO_PROJECT-clearsessions` with the following contents:

```
#!/bin/bash
export PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT
export DJANGO_SETTINGS_MODULE=settings
su $DJANGO_USER -c "/opt/$DJANGO_PROJECT/venv/bin/python \
    /opt/$DJANGO_PROJECT/manage.py clearsessions"
```

Make the file executable:

```
chmod 755 /etc/cron.daily/$DJANGO_PROJECT-clearsessions
```

In Unix-like systems, cron is the standard scheduler; it executes tasks at specified times. Scripts in `/etc/cron.daily` are executed once daily, starting at 06:25 (am) local time. The time to which this actually refers depends on the system's time zone, which you can find by examining the contents of the file `/etc/timezone`. In most of my servers, I use UTC. The time during which these scripts are run doesn't really matter much, but it's better to do it when the system is not very busy—especially if some of the scripts are intensive, such as backup (which we will see in a later chapter). For time zones with a positive UTC offset, 06:25 UTC could be a busy time, so you might want to change the system time zone with this command:

```
dpkg-reconfigure tzdata
```

There is a way to tell cron exactly at what time you want a task to run, but I won't go into that as throwing stuff into `/etc/cron.daily` should be sufficient for most use cases.

Cron expects all the programs it runs to be silent, i.e., to not display any output. If they do display output, cron emails that output to the administrator. This is very neat, because if your tasks only display output when there is an error, you will be emailed only when there is an error. However, for this to work, you must setup a local mail server as explained in *Using a local mail server*.

7.9 Chapter summary

- Install `dma` and (in the virtualenv) `django-sendmail-backend`
- Make sure `/etc/dma/dma.conf` has these contents:

  ```
  SMARTHOST $EMAIL_HOST
  PORT 587
  AUTHPATH /etc/dma/auth.conf
  SECURETRANSFER
  STARTTLS
  MAILNAME /etc/mailname
  ```

  Also make sure `/etc/dma/auth.conf` has these contents:

  ```
  $EMAIL_HOST_USER|$EMAIL_HOST:$EMAIL_HOST_PASSWORD
  ```

  Make sure `/etc/mailname` contains $DOMAIN.

- Create the cache directory:

```
mkdir /var/cache/$DJANGO_PROJECT/cache
chown $DJANGO_USER /var/cache/$DJANGO_PROJECT/cache
```

- Create file `/etc/cron.daily/$DJANGO_PROJECT-clearsessions` with the following contents:

```
#!/bin/bash
export PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_
 ↪PROJECT
export DJANGO_SETTINGS_MODULE=settings
su $DJANGO_USER -c "/opt/$DJANGO_PROJECT/venv/bin/python \
    /opt/$DJANGO_PROJECT/manage.py clearsessions"
```

Make the file executable:

```
chmod 755 /etc/cron.daily/$DJANGO_PROJECT-clearsessions
```

- Finally, this is the whole `settings.py` file:

```
from django_project.settings import *

debug = false
allowed_hosts = ['$domain', 'www.$domain']
databases = {
    'default': {
        'engine': 'django.db.backends.sqlite3',
        'name': '/var/opt/$django_project/$django_project.
↪db',
    }
}

server_email = 'noreply@$domain'
default_from_email = 'noreply@$domain'
admins = [
    ('$admin_name', '$admin_email_address'),
]
managers = admins
```

```
email_backend = 'django_sendmail_backend.backends.' \
                'emailbackend'

logging = {
    'version': 1,
    'disable_existing_loggers': false,
    'formatters': {
        'default': {
            'format': '[%(asctime)s] %(levelname)s: '
                      '%(message)s',
        }
    },
    'handlers': {
        'file': {
            'class': 'logging.timedrotatingfilehandler',
            'filename': '/var/log/$django_project/'
                        '$django_project.log',
            'when': 'midnight',
            'backupcount': 60,
            'formatter': 'default',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'info',
    },
}

caches = {
    'default': {
        'backend': 'django.core.cache.backends.filebased.'
                   'filebasedcache',
        'location': '/var/cache/$django_project/cache',
    }
}
```

CHAPTER

EIGHT

POSTGRESQL

8.1 Why PostgreSQL?

So far we have been using SQLite. Can we continue to do so? The answer, as always, is "it depends". Most probably you can't.

I'm using SQLite in production in one application I've made for an eshop hosted by BigCommerce. It gets the orders from the BigCommerce API and formats them on a PDF for printing on labels. It has no models, and all the data is stored in BigCommerce. The only significant data stored in SQLite is the users' names and passwords used for login, by `django.contrib.auth`. It's hardly three users. Recreating them would be easier than maintaining a PostgreSQL installation. So SQLite it is.

What if your database is small and you don't have many users, but you store mission-critical data in the database? That's a hard one. The thing is, no-one really knows if SQLite is appropriate, because no-one is using it for mission-critical data. Thunderbird doesn't use it for storing emails, but for storing indexes, which can be recreated. Likewise for Firefox. The SQLite people claim it's appropriate for mission-critical applications, but industry experience on that is practically nonexistent. I've never seen corruption in SQLite. I've seen corruption in PostgreSQL, but we are comparing apples to

oranges. I have a gut feeling (but no hard data) that I can trust SQLite more than MySQL.

If I ever choose to use SQLite for mission-critical data, I will make sure I not just backup the database file, but also backup a plain text dump of the database. I trust plain text dumps more than database files in case there is silent corruption that can go unnoticed for some time.

One problem with SQLite is that you may choose to go with it now that your database is small and your users are few, but you can't really be certain what it will be like in three or five years. If for some reason the database has grown or the users have increased, SQLite might be unable to handle it. Migrating to PostgreSQL at that stage could be a nightmare. So the safe option is to use PostgreSQL straight from the beginning.

As for MySQL, I never understood why it has become so popular when there's PostgreSQL around. My only explanation is it was marketed better. PostgreSQL is more powerful, it is easier, and it has better documentation. If you have a reason to use MySQL, it's probably that you already know it, or that people around you know it (e.g. it is company policy). In that case, hopefully you don't need any help from me. Otherwise, choose PostgreSQL and read the rest of this chapter.

8.2 Getting started with PostgreSQL

You may have noticed that I prefer to tell you to do things first and then explain them. Same thing again. We will quickly install PostgreSQL and configure Django to use it. You won't be understanding clearly what you are doing. After we finish it, you have some long sections to read. You *must* read them, however. **The way to avoid doing the reading is to forget about PostgreSQL and continue using SQLite.** It is risky to put your customer's data on a system that you don't understand and that you've set up just by blindly following instructions.

```
apt install postgresql
```

This will install PostgreSQL and create a cluster; I will explain later what this means.

Warning: Make sure the locale is right

When PostgreSQL installs, it uses the encoding specified by the default system locale (found in `/etc/default/locale`). If this is not UTF-8, the databases will be using an encoding other than UTF-8. You really don't want that. If you aren't certain, you can check, using the procedure I explained in *Setting up the system locale*, that the default system locale is appropriate. You can also check that PostgreSQL was installed with the correct locale with this command:

```
su postgres -c 'psql -l'
```

This will list your databases and some information about them, including their locale. Immediately after installation, there should be three databases (I explain them later on).

If you make an error and install PostgreSQL while the locale is wrong, the easiest way to fix the problem is to drop and recreate the cluster. I explain later what "cluster" means, but what you need to know is that the following procedure will permanently and irrevocably delete all your databases. **Be careful not to type the commands in the wrong window** (you could delete the databases of the wrong server). Fix your locale as described in *Setting up the system locale*, then execute the following commands:

```
service postgresql stop
pg_dropcluster 9.5 main
pg_createcluster 9.5 main
service postgresql start
```

If you have a database with useful data, obviously you can't do this. Fix-

> ing the problem is more advanced and isn't covered by this chapter; there is a question at Stackoverflow that treats it, but better finish this chapter first to get a grip on the basics.

Let's now try to connect to PostgreSQL with a client program:

```
su postgres -c 'psql template1'
```

This connects you with the "template1" database and gives you a prompt ending in `#`. You can give it some commands like `\l` to list the databases (there are three just after installation). Let's create a user and a database. I will use placeholders $DJANGO_DB_USER, $DJANGO_DB_PASSWORD, and $DJANGO_DATABASE. We normally use the same as $DJANGO_PROJECT for both $DJANGO_DB_USER and $DJANGO_DATABASE, and I have the habit of using the SECRET_KEY as the database password, but in principle all these can be different; so I will be using these different placeholders here to signal to you that they denote something different.

```
CREATE USER $DJANGO_DB_USER PASSWORD '$DJANGO_DB_PASSWORD';
CREATE DATABASE $DJANGO_DATABASE OWNER $DJANGO_DB_USER;
```

The command to exit `psql` is `\q`.

Next, we need to install `psycopg2`:

```
apt install python-psycopg2 python3-psycopg2
```

This will work only if you have created your virtualenv with the `--system-site-packages` option, which is what I told you to do many pages ago. Otherwise, you need to `pip install psycopg2` inside the virtualenv. Most people do it in the second way. However, attempting to install `psycopg2` with `pip` will require compilation, and compilation can be tricky, and different `psycopg2` versions might behave differently, and in my experience the easiest and safest way is to install the version of `psycopg2` that is packaged with the

operating system. If your site-wide Python installation is clean (meaning you have used `pip` only in virtualenvs), `--system-site-packages` works great.

Finally, change your `DATABASES` setting to this:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': '$DJANGO_DATABASE',
        'USER': '$DJANGO_DB_USER',
        'PASSWORD': '$DJANGO_DB_PASSWORD',
        'HOST': 'localhost',
        'PORT': 5432,
    }
}
```

From now on, Django should be using PostgreSQL (you may need to restart Gunicorn). You should be able to setup your database with this:

```
PYTHONPATH=/etc/opt/$DJANGO_PROJECT:/opt/$DJANGO_PROJECT \
DJANGO_SETTINGS_MODULE=settings \
su $DJANGO_USER -c \
"/opt/$DJANGO_PROJECT/venv/bin/python \
/opt/$DJANGO_PROJECT/manage.py migrate"
```

8.3 PostgreSQL connections

A short while ago we run this innocent looking command:

```
su postgres -c 'psql template1'
```

Now let's explain what this does. Brace yourself, as it will take several sections. Better go make some tea, relax, and come back.

A web server listens on TCP port 80 and a client, usually a browser, connects

to that port and asks for some information. The server and the client communicate in a language, in this case the Hypertext Transfer Protocol or HTTP. In very much the same way, the PostgreSQL server is listening on a communication port and a client connects to that port. The client and the server communicate in the PostgreSQL Frontend/Backend Protocol.

In the case of the `psql template1` command, `psql`, the PostgreSQL interactive terminal, is the client. It connects to the server, and gets commands from you. If you tell it `\l`, it asks the server for the list of databases. If you give it an SQL command, it sends it to the server and gets the response from the server.

When you connect to a web server with your browser, you always provide the server address in the form of a URL. But here we only provided a database name. We could have told it the server as follows (but it's not going to work without a fight, because the user authentication kicks in, which I explain in the next section):

```
psql --host=localhost --port=5432 template1
```

You might think `localhost` and 5432 is the default, but it isn't. The default is Unix domain socket `/var/run/postgresql/.s.PGSQL.5432`. Let's see what this means.

If you think about it, TCP is nothing more than a way for different processes to communicate. One process, the browser, opens a communication channel to another process, the web server. Unix domain sockets are an alternative interprocess communication system that has some advantages but only works on the same machine. Two processes on the same machine that want to communicate can do so via a socket; one process, the server, will create the socket, and another, the client, will connect to the socket. One of the philosophies of Unix is that everything looks like a file, so Unix domain sockets look like files, but they don't occupy any space on your disk. The client opens what looks like a file, and sends and receives data from it.

When the PostgreSQL server starts, it creates socket `/var/run/postgresql/.s.PGSQL.5432`. The "5432" is noth-

ing of meaning to the system; if the socket had been named `/var/run/postgresql/hello.world`, it would have worked exactly the same. The PostgreSQL developers chose to include the "5432" in the name of the socket as a convenience, in order to signify that this socket leads to the same PostgreSQL server as the one listening on TCP port 5432. This is useful in the rare case where many PostgreSQL instances (called "clusters", which I explain later) are running on the same machine.

Hint: Hidden files

In Unix, when a file begins with a dot, it's "hidden". This means that `ls` doesn't normally show it, and that when you use wildcards such as `*` to denote all files, the shell will not include it. Otherwise it's not different from non-hidden files.

To list the contents of a directory including hidden files, use the `-a` option:

```
ls -a /var/run/postgresql
```

This will include `.` and `..`, which denote the directory itself and the parent directory (`/var/run/postgresql/.` is the same as `/var/run/postgresql`; `/var/run/postgresql/..` is the same as `/var/run`). You can use `-A` instead of `-a` to include all hidden files except `.` and `..`.

8.4 PostgreSQL roles and authentication

After a client such as `psql` connects to the TCP port or to the Unix domain socket of the PostgreSQL server, it must authenticate before doing anything else. It must login, so to speak, as a user. Like many other relational database management systems (RDBMS's), PostgreSQL keeps its own list of users and has a sophisticated permissions system with which different users have different permissions on different databases and tables. This is useful in desktop applications. In the Greek tax office, for example, employees run a program

on their computer, and the program asks them for their username and password, with which they login to the tax office RDBMS, which is Oracle, and Oracle decides what this user can or cannot access.

Web applications changed that. Instead of PostgreSQL managing the users and their permissions, we have a single PostgreSQL user, $DJANGO_DB_USER, as which Django connects to PostgreSQL, and this user has full permissions on the $DJANGO_DB database. The actual users and their permissions are managed by `django.contrib.auth`. What a user can or cannot do is decided by Django, not by PostgreSQL. This is a pity because `django.contrib.auth` (or the equivalent in other web frameworks) largely duplicates functionality that already exists in the RDBMS, and because having the RDBMS check the permissions is more robust and more secure. I believe that the reason web frameworks were developed this way is independence from any specific RDBMS, but I don't really know. Whatever the reason, we will live with that, but I am telling you the story so that you can understand why we need to create a PostgreSQL user for Django to connect to PostgreSQL as.

Just as in Unix the user "root" is the superuser, meaning it has full permissions, and likewise the "administrator" in Windows, in PostgreSQL the superuser is "postgres". I am talking about the database user, not the operating system user. There is also an operating system "postgres" user, but here I don't mean the user that is stored in `/etc/passwd` and which you can give as an argument to `su`; I mean a PostgreSQL user. The fact that there exists an operating system user that happens to have the same username is irrelevant.

Let's go back to our innocent looking command:

```
su postgres -c 'psql template1'
```

As I explained, since we don't specify the database server, `psql` by default connects to the Unix domain socket `/var/run/postgresql/.s.PGSQL.5432`. The first thing it must do after connecting is authenticating. We could have specified a user to authenticate as with the `--username` option. Since we did not, `psql` uses the default. The default is what the `PGUSER` environment

variable says, and if this is absent, it is the username of the current operating system user. In our case, the operating system user is `postgres`, because we executed `su postgres`; so `psql` attempts to authenticate as the PostgreSQL user `postgres`.

To make sure you understand this clearly, try to run `psql template1` as root:

```
psql template1
```

What does it tell you? Can you understand why? If not, please re-read the previous paragraph. Note that after you have just installed PostgreSQL, it has only one user, `postgres`.

So, `psql` connected to `/var/run/postgresql/.s.PGSQL.5432` and asked to authenticate as `postgres`. At this point, you might have expected the server to request a password, which it didn't. The reason is that PostgreSQL supports many different authentication methods, and password authentication is only one of them. In that case, it used another method, "peer authentication". By default, PostgreSQL is configured to use peer authentication when the connection is local (that is, through the Unix domain socket) and password authentication when the connection is through TCP. So try this instead to see that it will ask for a password:

```
su postgres -c 'psql --host=localhost template1'
```

You don't know the `postgres` password, so just provide an empty password and see that it refuses the connection. I don't know the password either. I believe that Debian/Ubuntu sets no password (i.e. invalid password) at installation time. You can set a valid password with `ALTER USER postgres PASSWORD 'topsecret'`, but don't do that. There is no reason for the `postgres` user to connect to the database with password authentication, it could be a security risk, and you certainly don't want to add yet another password to your password manager.

Let's go back to what we were saying. `psql` connected to the socket and asked to authenticate as `postgres`. The server decided to use peer authentication,

because the connection is local. In peer authentication, the server asks the operating system: "who is the user who connected to the socket?" The operating system replied: "postgres". The server checks that the operating system user name is the same as the PostgreSQL user name which the client has requested to authenticate as. If it is, the server allows. So the Unix `postgres` user can always connect locally (through the socket) as the PostgreSQL `postgres` user, and the Unix `joe` user can always connect locally as the PostgreSQL `joe` user.

So, in fact, if $DJANGO_USER and $DJANGO_DB_USER are the same (and they are if so far you have followed everything I said), you could use these Django settings:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': '$DJANGO_DATABASE',
        'USER': '$DJANGO_DB_USER',
    }
}
```

In this case, Django will connect to PostgreSQL using the Unix domain socket, and PostgreSQL will authenticate it with peer authentication. This is quite cool, because you don't need to manage yet another password. However, I don't recommend it. First, most of your colleagues will have trouble understanding that setup, and you can't expect everyone to sit down and read everything and understand everything in detail. Second, next month you may decide to put Django and PostgreSQL on different machines, and using password authentication you make your Django settings ready for that change. It's also better, both for automation and your sanity, to have similar Django settings on all your deployments, and not to make some of them different just because it happens that PostgreSQL and Django run on the same machine there.

Remember that when we created the $DJANGO_DATABASE database, we made $DJANGO_DB_USER its owner?

```
CREATE DATABASE $DJANGO_DATABASE OWNER $DJANGO_DB_USER;
```

The owner of a database has full permission to do anything in that database: create and drop tables; update, insert and delete any rows from any tables; grant other users permission to do these things; and drop the entire database. This is by far the easiest and recommended way to give $DJANGO_DB_USER the required permissions.

Before I move to the next section, two more things you need to know. PostgreSQL authentication is configurable. The configuration is at `/etc/postgresql/9.x/main/pg_hba.conf`. Avoid touching it, as it is a bit complicated. The default (peer authentication for Unix domain socket connections, password authentication for TCP connections) works fine for most cases. The only problem you are likely to face is that the default configuration does not allow connection from other machines, only from localhost. So if you ever put PostgreSQL on a different machine from Django, you will need to modify the configuration.

Finally, PostgreSQL used to have users and groups, but the PostgreSQL developers found out that these two types of entity had so much in common that they joined them into a single type that is called "role". A role can be a member of another role, just as a user could belong to a group. This is why you will see "role joe does not exist" in error messages, and why `CREATE USER` and `CREATE ROLE` are exactly the same thing.

8.5 PostgreSQL databases and clusters

Several pages ago, we gave this command:

```
su postgres -c 'psql template1'
```

I have explained where it connected and how it authenticated, and to finish this up I only need to explain why we told it to connect to the "template1" database.

The thing is, there was actually no theoretical need to connect to a database. The only two commands we gave it were these:

```
CREATE USER $DJANGO_DB_USER PASSWORD '$DJANGO_DB_PASSWORD';
CREATE DATABASE $DJANGO_DATABASE OWNER $DJANGO_DB_USER;
```

I also told you, for experiment, to also provide the `\l` command, which lists the databases.

All three commands are independent of database and would work exactly the same regardless of which database we are connected to. However, whenever a client connects to PostgreSQL, it *must* connect to a database. There is no way to tell the server "hello, I'm user postgres, authenticate me, but I don't want to connect to any specific database because I only want to do work that is independent of any specific database". Since you must connect to a database, you can choose any of the three that are always known to exist: `postgres`, `template0`, and `template1`. It is a long held custom to connect to `template1` in such cases (although `postgres` is a bit better, but more on that below).

The official PostgreSQL documentation explains `template0` and `template1` so perfectly that I will simply copy it here:

> CREATE DATABASE actually works by copying an existing database. By default, it copies the standard system database named `template1`. Thus that database is the "template" from which new databases are made. If you add objects to `template1`, these objects will be copied into subsequently created user databases. This behavior allows site-local modifications to the standard set of objects in databases. For example, if you install the procedural language PL/Perl in `template1`, it will automatically be available in user databases without any extra action being taken when those databases are created.
>
> There is a second standard system database named `template0`. This database contains the same data as the initial contents of `template1`, that is, only the standard objects predefined by your version of PostgreSQL. `template0` should never be changed af-

> ter the database cluster has been initialized. By instructing CREATE DATABASE to copy `template0` instead of `template1`, you can create a "virgin" user database that contains none of the site-local additions in `template1`. This is particularly handy when restoring a `pg_dump` dump: the dump script should be restored in a virgin database to ensure that one recreates the correct contents of the dumped database, without conflicting with objects that might have been added to `template1` later on.

There's more about that in Section 22.3 of the documentation. In practice, I never touch `template1` either. I like to have PostGIS in the template, but what I do is create another template, `template_postgis`, for the purpose.

Before explaining what the `postgres` database is for, we need to look at an alternative way of creating users and databases. Instead of using `psql` and executing `CREATE USER` and `CREATE DATABASE`, you can run these commands:

```
su postgres -c "createuser --pwprompt $DJANGO_DB_USER"
su postgres -c "createdb --owner=$DJANGO_DB_USER $DJANGO_DATABASE
 ↪"
```

Like `psql`, `createuser` and `createdb` are PostgreSQL clients; they do nothing more than connect to the PostgreSQL server, construct `CREATE USER` and `CREATE DATABASE` commands from the arguments you have given, and send these commands to the server. As I've explained, whenever a client connects to PostgreSQL, it *must* connect to a database. What `createuser` and `createdb` (and other PostgreSQL utility programs) do is connect to the `postgres` database. So `postgres` is actually an empty, dummy database used when a client needs to connect to the PostgreSQL server without caring about the database.

I hinted above that it is better to use `psql postgres` than `psql template1` (though most people use the latter). The reason is that sometimes you may accidentally create tables while being connected to the wrong database. It has happened to me more than once to screw up my `template1` database. You don't want to accidentally modify your `template1` database, but it's not a big

deal if you modify your `postgres` database. So use that one instead when you want to connect with `psql`. The only reason I so far told you to use the suboptimal `psql template1` is that I thought you would be confused by the many instances of "postgres" (there's an operating system user, a PostgreSQL user, and a database named thus).

Now let's finally explain what a cluster is. Let's see it with an example. Remember that nginx reads `/etc/nginx/nginx.conf` and listens on port 80? Well, it's entirely possible to start another instance of nginx on the same server, that reads `/home/antonis/nginx.conf` and listens to another port. That other instance will have different lock files, different log files, different configuration files, and can have different directory roots, so it can be totally independent. It's very rarely needed, but it can be done (I've done it once to debug a production server of a problem I couldn't reproduce in development). Likewise, you can start a second instance of PostgreSQL, that uses different configuration files and a different data file directory, and listens on a different port (and different Unix domain socket). Since it is totally independent of the other instance, it also has its own users and its own databases, and is served by different server processes. These server processes could even be run by different operating system users (but in practice we use the same user, `postgres`, for all of them). Each such instance of PostgreSQL is called a cluster. By far most PostgreSQL installations have a single cluster called "main", so you needn't worry further about it; just be aware that this is why the configuration files are in `/etc/postgresql/9.x/main`, why the data files are in `/var/lib/postgresql/9.x/main`, and why the log files are named `/var/log/postgresql/postgresql-9.x-main.log`. If you ever create a second cluster on the same machine, you will be doing something advanced, like setting up certain kinds of replication. If you are doing such an advanced thing now, you are probably reading the wrong book.

8.6 Further reading

You may have noticed that I close most chapters with a summary, which, among other things, repeats most of the code and configuration snippets of the chapter. In this chapter I have no summary to write, because I have already written it; it's Section *Getting started with PostgreSQL*. In the rest of the chapter I merely explained it.

I explain in the next chapter, but it is so important that I must repeat it here, that **you should not backup your PostgreSQL database by copying its data files from /var/lib/postgresql**. If you do such a thing, you risk being unable to restore it when you need it. Read the next chapter for more information.

I hope I wrote enough to get you started. You should be able to use it in production now, and learn a little bit more and more as you go on. Its great documentation is the natural place to continue. If you ever do anything advanced, Gregory Smith's PostgreSQL High Performance is a nice book.

CHAPTER

NINE

RECOVERY PART 1

9.1 Why "recovery"?

Usually book chapters and blog posts dealing with what I'm dealing in this chapter call it "backup and recovery". To me, backup is just a part of recovery, it is only the first step towards recovery. This is why I prefer to just use "recovery". It's not just a language issue, it's a different way of thinking. When you deal with "backup and recovery", you view them as two separate things. You might finish your backup and think "I'm through with this, I'll deal with recovery if and when the time comes". When we name it just "recovery", you understand that backup isn't something isolated, and certainly it isn't the point. Backup on its own is useless and pointless. Your customer doesn't care about backup; they care about whether you are able to recover the system when it breaks. In fact, they don't even care about that; they just care that the system works, and they prefer to not know what you are doing behind the scenes for it to work. One of the things you are doing behind the scenes is to recover the system.

The most important thing about recovery is that it should be tested. Once a year, or once in two years, you should switch off the server, pretend it exploded, and recover on a new server. Without doing this, you will not know if you can recover. Recovery plans contain all sorts of silly errors.

Maybe your backups are encrypted, and the decryption key is only stored in the server itself, and you won't have it when you need to recover. Maybe you don't backup some files that you think can be recreated, and it turns that among them there are some valuable data files. The thing is, you won't be able to know what you did wrong until you test your recovery.

Untested recovery always takes way longer than you think. When you have written down the recovery procedure and you have tested it, you may be able to recover within a couple of hours or even a few minutes, with minimal stress. It can be part of your day-to-day work and not a huge event. Without a written procedure, or with an untested procedure, you will be sweating over your keyboard for a whole day or more, while your customer will be frustrated. It's hard to imagine how much time you can waste because you are getting a `pg_restore` option wrong until you try it.

So, **think about recovery. Recovery starts with backup, continues with the creation of a written restore procedure, and is mostly completed when that procedure is tested.** Anything less than that is dangerous.

9.2 Where to backup

The cloud is very attractive. Amazon, Google, Backblaze, Microsoft, they sell cheap storage. All your server has to do is save its stuff there. You don't need to change tapes every day and move them off site, as we used to do 10 years ago. And your backup is on another continent. No chance it will explode the same time as your server, right? Wrong!

The problem is that your system has a single point of failure: the security of your server. For your server to backup itself to the remote storage, it must have write access to the remote storage. So if the security of your server is compromised, the attacker can delete your server's data *and* the backup.

Do you think this is far-fetched? Code Spaces was a company that had its code and data on Amazon Web Services. One day in 2014 an attacker managed to get access to their account and demanded ransom. Negotiations didn't

go well and the attacker deleted all data. All backups. Everything. The company was wiped out overnight. It ceased to exist. Undoubtedly its customers were also damaged.

Forget about two-factor authentication or Amazon's undeletable S3 files. Your phone might be stolen. Or the employee who has access to the account *and* has two factor-authentication on his phone might go crazy and want to harm you. Or *you* might go crazy and want to hurt your customers. Or you might be paying the server and the backup from the same credit card, with the same obsolete email in both, and the credit card might be cancelled, and you'd fail to receive the emails, and the providers might delete the server and the backups at the same time. Or the whole system, regardless its safety checks and everything, might have a bug somewhere. Our experience of the last 20 years does not indicate that systems are getting safer; on the contrary. Heartbleed and Shellshock showed how vulnerable the whole Internet is; and the next such problem is just waiting to be discovered.

The only way to be reasonably certain that your data is safe is if the backup is offline, on a medium you can actually touch, disconnected from the network. But this is very expensive, so you need to compromise on something.

What I do is backup my systems online daily, but I also copy the backup to a disk once a month, and I take the disk offline. The next month I use another disk. I will tell you more about it later on.

9.3 Estimating storage cost

Cloud storage services advertise a cost per GB per month. For example, for Backblaze the amount at the time of this writing is $0.005. We need to multiply this by 12 to arrive at a cost of $0.06 per year.

Depending on the backup scheme you use, you might save the data multiple times. For example, the scheme I will propose involves a full backup every three months, and backups kept for two years. This means that each GB will

be stored a total of eight times. So this means that each GB of data, or eight GB of backup storage, will cost $0.48 per year.

There are also charges for downloading. Backblaze charges $0.05 per GB for each download. If you download the backups twice a year for recovery testing, that's $0.10. So the total so far is $0.58 per GB per year. For a Django installation with 10 GB of data, this will be $5.80 per year. For 30 GB of data, it will be $17.40 per year. While it is not much, if you maintain many Django installations it can add up, so you must make sure you take the cost into account when you offer a support contract to the customer.

If you download the backups once a month in order to save them to an offline disk, this will cost an additional $0.05 per month, which amounts to $0.60 per year, so this doubles online storage costs. In the scheme I explain in the next chapter, we take offline backups directly from the server, not from the online backups, so you don't have this cost. However, it's perfectly valid to backup the backups instead, and sometimes it's preferable; if you do it this way, don't forget to take the download cost into account.

If you use external disks for offline backups, you need two disks, and each disk must have a capacity of all the data of all your installations combined. They must be rotating disks (i.e. not SSD), preferably portable USB ones. You may also be able to use SATA disks with a SATA-to-USB adapter; however, one of the advantages of USB disks is that it's much easier to label them by attaching a sticker (SATA disks have very little space available for attaching a sticker, unless you cover their original label, which you don't want). You might want to use small (2.5-inch) disks, which are much easier to carry. In any case, in this book we deal with deployments on a single server, so these are probably small and a 1 TB disk is likely enough for all your deployments. Two such external disks cost around $100. They might live for five years, but I prefer to be more conservative and assume they'll last for a maximum of two years; your backup schemes, your customers, and your business in general will have changed enough by then. So the total cost of backup (assuming it all fits in a 1 TB disk) is $50 per year plus $0.58 per GB per year.

9.4 Setting up backup storage

How exactly you will setup your backup storage depends on the type of storage you use. You might use Backblaze B2, Google Cloud Storage, Amazon S3, or various other services. If you have a static IP address, you could also setup a physical machine, but this is typically harder and more expensive. In the rest of this chapter, I will assume you are using Backblaze B2. If you are familiar with another storage system, go ahead and use that. (Note: I am not affiliated with Backblaze.)

To setup your backup storage on Backblaze, go to https://backblaze.com/, select "B2 Cloud Storage", and sign up or login. Then create a bucket.

A bucket is a virtual hard disk, so to speak. It has no fixed size; it grows as you add files to it. Rather than having different buckets for different customers, in this chapter I assume you have only one bucket, which is simpler. Remember, always choose the simplest solution first, and don't make assumptions about how the future will be; very often you ain't gonna need it. If and when the future brings in needs that can't be covered by the solution I'm proposing here, you will need to revise your strategy.

In order to create the bucket, you will be asked for a name, and about whether it's going to be private or public. It will be private of course; as for the name, I like `$NICK-backup`, where `$NICK` is my usual username (such as the one you have on Twitter perhaps). After you create it, go to the Bucket Settings, and tell it to keep only the last version of the file versions. This is because whenever you change a file, or whenever you delete a file, Backblaze B2 has the option of also keeping the previous version of the file. While this can be neat in some use cases, we won't be needing it here and it's going to be a waste of disk space (and therefore money). We just want the bucket to behave like a normal hard disk.

Now, if you go to the "Buckets" section of the Backblaze B2 dashboard ("Buckets" is actually the front page of the dashboard), near the top it says "Show Accout ID and Application Key". Click on that link and it will show you your Account ID. If you don't know your Application Key (for example,

if it's your first time in Backblaze B2) create a new one. Take note of both your Account ID and your Application Key; we will need them later. I will be calling them $ACC_ID and $APP_KEY.

9.5 Setting up duplicity and duply

The recovery software we will use is duplicity. While it works quite well, it is hard to use on its own because its user interface is inconvenient. It does not have a configuration file, but you tell it everything it needs to know on the command line, and a very long command line indeed. I believe that the authors of duplicity intended it to be run by scripts and not by humans. Here we are going to use duply, a front-end to duplicity that makes our job much easier. Let's start by installing it:

```
apt install duply
```

Hint: Installing duplicity in Debian

Although `apt install duply` will work on Debian 8, it will install duplicity 0.6.24, which does not support Backblaze B2. Therefore, you may want to install a more recent version of duplicity.

Go to duplicity's home page, http://duplicity.nongnu.org/, and copy the link to the current release in the Download section. I will call it $DUPLICITY_TARBALL_SOURCE, and I will also use the placeholder $DUPLICITY_VERSION.

Install duplicity with the following commands:

```
apt install python-dev build-essential \
    python-setuptools librsync-dev
cd
wget $DUPLICITY_TARBALL_SOURCE
tar xzf duplicity-$DUPLICITY_VERSION.tar.gz
cd duplicity-$DUPLICITY_VERSION
python setup.py install
```

`wget` downloads stuff from the web. You give it a URL, it fetches it and stores it in a file. In this case, it will fetch file `duplicity-$DUPLICITY_VERSION.tar.gz` and store it in the current directory (which should be `/root` if you run `cd` as I suggested).

`tar` is very roughly the equivalent of `zip`/`unzip` on Unix; it can create and read files containing other files (but `tar` can't read zip files, neither can `zip` read tar files). These files are called "archive files". The `x` in `xzf` means that the desired operation is extraction of files from an archive (as opposed to `c`, which is the creation of an archive, or `t`, which is for listing the contents of an archive); the `z` means that the archive is compressed; and `f` means that "the next argument in the command line is the archive name". I have long forgotten what it does if you don't specify the `f` option, but the default was something suitable for 1979, when the first version of `tar` was created and had to do with tape drives (in fact "tar" is short for "tape archiver"). If more arguments follow, they are names of files to extract from the archive. Since we don't specify any, it will extract all files. In this particular archive, all contained files are in directory `duplicity-$DUPLICITY_VERSION`, so `tar` creates the directory to put the files in there.

Next, let's create a configuration directory:

```
mkdir -p /etc/duply/main
chmod 700 /etc/duply/main
```

With duply you can create many different configurations which it calls "profiles". We only need one here, and we will call it "main". This is why we created directory `/etc/duply/main`. Inside it, create a file called `conf`, with the following contents:

```
GPG_KEY=disabled

SOURCE=/
```

```
TARGET=b2://$ACC_ID:$APP_KEY@$NICK-backup/$SERVER_NAME/

MAX_AGE=2Y
MAX_FULLS_WITH_INCRS=2
MAX_FULLBKP_AGE=3M
DUPL_PARAMS="$DUPL_PARAMS --full-if-older-than $MAX_FULLBKP_AGE "

VERBOSITY=warning
ARCH_DIR=/var/cache/duplicity/duply_main/
```

Warning: Syntax is bash

The duply configuration file is neither Python (such as `settings.py`) nor an ini-style file; it is a shell script. This notably means that, when defining variables, there can be no space on either side of the equals sign ('='). Strings need to be quoted only if they contain spaces, so, for example, the following three definitions are exactly the same:

```
GREETING=hello
GREETING="hello"
GREETING='hello'
```

However, variables are replaced inside double quotes, but not inside single quotes:

```
WHO=world
GREETING1="hello, $WHO"
GREETING2='hello, $WHO'
```

After this is run, `GREETING1` will have the value "hello, world", whereas `GREETING2` will be "hello, $WHO". You can experiment by simply typing these commands in the shell prompt, and examine the values of variables with `echo $GREETING1` and so on.

Also create a file `/etc/duply/main/exclude`, with the following contents:

```
- /dev
- /proc
- /sys
- /run
- /var/lock
- /var/run
- /lost+found
- /boot
- /tmp
- /var/tmp
- /media
- /mnt
- /var/cache
- /var/crash
- /var/swap
- /var/swapfile
- /var/swap.img
- /var/lib/mysql
- /var/lib/postgresql
```

You can now backup your system by executing this command:

```
duply main backup
```

If this is a small virtual server, it should finish in a few minutes. **This, however, is just a temporary test.** There are many things that won't work correctly, and one of the most important is that we haven't backed up PostgreSQL (and MySQL, if you happen to use it), and any SQLite files we backed up may be corrupted. We just made this test to get you up and running. Let me now explain what these configuration files mean.

9.6 Duply configuration

Let's check again the duply configuration file, `/etc/duply/main/conf`:

```
GPG_KEY=disabled

SOURCE=/
TARGET=b2://$ACC_ID:$APP_KEY@$NICK-backup/$SERVER_NAME/

MAX_AGE=2Y
MAX_FULLS_WITH_INCRS=2
MAX_FULLBKP_AGE=3M
DUPL_PARAMS="$DUPL_PARAMS --full-if-older-than $MAX_FULLBKP_AGE "

VERBOSITY=warning
ARCH_DIR=/var/cache/duplicity/duply_main/
```

GPG_KEY=disabled Duplicity, and therefore duply, can encrypt the backups. The rationale is that the backup storage provider shouldn't be able to read your files. So if you have a company, and you have a server at the company premises, and you backup the server at Backblaze or at Google, you might not want Backblaze or Google to be able to read the company's files. In our case this would achieve much less. Our virtual server provider can read our files anyway, since they are stored in our virtual server, in a data centre owned by the provider. Making it impossible for Backblaze to read our files doesn't achieve much if DigitalOcean can read them. Encrypting the backups is often more trouble than what it's worth, so we just disable it.

SOURCE=/ This specifies the directory to backup. We specify the root directory in order to backup the entire file system. We will actually exclude some files and directories as I explain in the next section.

TARGET=b2://... This is the place to backup to. The first part, `b2:`, specifies the "storage backend". Duplicity supports many storage backends; they are listed in `man duplicity`, Section "URL Format". As you can see, the syntax for the Backblaze B2 backend is "b2://account_id:application_key@bucket/directory". Even if you have only one server, it's likely that soon you will have more, so store your backups in the $SERVER_NAME directory.

MAX_AGE=2Y This means that backups older than 2 years will be deleted. Note that, if your databases and files contain customer data, it may be illegal to keep the backups for more than a specified amount of time. If a user decides to unsubscribe or otherwise remove their data from your database, you are often required to delete every trace of your customer's data from everywhere, including the backups, within a specified amount of time, such as six months or two years. You need to check your local privacy laws.

MAX_FULLS_WITH_INCRS=2, MAX_FULLBKP_AGE=3M A **full backup** backs up everything. In an **incremental backup** only the things that have changed since the previous backup are backed up. So if on 12 January you perform a full backup, an incremental backup on 13 January will only save the things that have changed since 12 January, and another incremental on 14 January will only save what has changed since 13 January. `MAX_FULLBKP_AGE=3M` means that every three months a new full backup will occur. `MAX_FULLS_WITH_INCRS=2` means that incremental backups will be kept only for the last two full backups; for older full backups, incrementals will be removed.

Collectively these parameters (together with `MAX_AGE=2Y`) mean that a total of about eight full backups will be kept; for the most recent three to six months, the daily history of the files will be kept, whereas for older backups the quarterly history will be kept. You will thus be able to restore your system to the state it was two days ago, or three days ago, or 58 days ago, but not necessarily exactly 407 days ago—you will need to round this up to about 45 days earlier or later.

Keeping the history of your system is very important. It is common to lose some data and realize it some time later. If each backup simply overwrote the previous one, and you realized today that you had accidentally deleted a file four days ago, you'd be in trouble.

DUPL_PARAMS="$DUPL_PARAMS ..." If you want to add any parameters to duplicity that have not been foreseen in duply, you can specify them in `DUPL_PARAMS`. Duply just takes the value of `DUPL_PARAMS` and

adds it to the duplicity command line. Duply does not directly support `MAX_FULLBKP_AGE`, so we need to manually add it to `DUPL_PARAMS`.

The `$DUPL_PARAMS` and `$MAX_FULLBKP_AGE` should be included literally in the file, the aren't placeholders such as `$NICK`, `$ACC_ID` and `$APP_KEY`

VERBOSITY=warning Options are error, warning, notice, info, and debug. "warning" will show warnings and errors; "notice" will show notices and warnings and errors; and so on. "warning" is usually fine.

ARCH_DIR=/var/cache/duplicity/duply_main/ Duplicity keeps a cache on the local machine that helps it know what things it has backed up, without actually needing to fetch that information from the backup storage—this speeds things up and lessens network traffic. If this local cache is deleted, it recreates it by reading stuff from remotely. Duply's default cache path is suboptimal so we change it.

In order to see duply's documentation for these settings you need to ask it to create a configuration file. We created the configuration files above ourselves, but we could have given the command `duply main create`, and this would have created `/etc/duply/main/conf` and `/etc/duply/main/exclude`; actually it creates these files under `/etc/duply` only if that directory exists; otherwise it creates them under `~/.duply`. After it creates the files, you are supposed to go and edit them. The automatically created `conf` is heavily commented and the comments explain what each setting does. So if you want to read the docs, `duply tmp create`, then go to `/etc/duply/tmp/conf` and read.

When you run duply what it actually does is read your configuration files, convert them into command line arguments for duplicity, and execute duplicity with a huge command line. For this reason, the documentation of duply's settings often refers you to duplicity. For example, for details on `MAX_FULLS_WITH_INCRS`, the comments in `conf` tell you to execute `man duplicity` and read about `remove-all-inc-of-but-n-full`.

9.7 Excluding files

The file `/etc/duply/main/exclude` contains files and directories that shall be excluded from the backup. Actually it uses a slightly complicated language that allows you to say things like "exclude directory X but include X/Y but do not include X/Y/Z". However, we will use it in a simple way, just in order to exclude files and directories, which means we just precede each path with "-". The exclude file we specified two sections ago is this:

```
- /dev
- /proc
- /sys
- /run
- /var/lock
- /var/run
- /lost+found
- /boot
- /tmp
- /var/tmp
- /media
- /mnt
- /var/cache
- /var/crash
- /var/swap
- /var/swapfile
- /var/swap.img
- /var/lib/mysql
- /var/lib/postgresql
```

/dev, /proc, /sys In these directories you will not find real files. `/dev` contains device files. In Unix most devices look like files. In fact, one of the Unix principles is that everything is a file. So the first hard disk is usually `/dev/sda` (but in virtual machines it is often `/dev/vda`). `/dev/sda1` (or `/dev/vda1`) is the first partition of that disk. You can actually open `/dev/sda` (or `/dev/vda`) and write to it (the root user has permission to do so), which will of course corrupt your system. Reading it is not a

problem though (but it's rarely useful).

`/sys` and `/proc` contain information about the system. For example, `/proc/meminfo` contains information about RAM, and `/proc/cpuinfo` about the CPU. You can examine the contents of these "files" by typing, for example, `cat /proc/meminfo` (`cat` prints the contents of files).

The `/dev`, `/sys` and `/proc` directories exist on your disk only as empty directories. The "files" inside them are created by the kernel, and they do not exist on the disk. Not only does it not make sense to backup, you would also be in trouble if you attempted to.

/run, /var/lock, /var/run `/run` stores information about running services, in order to keep track of them. This information is mostly process ids and locks. For example, `/run/sshd.pid` contains the process id of the SSH server. The system will use this information if, for example, you ask to restart the SSH server. Whenever the system boots, it empties that directory, otherwise the system would be confused. In older versions such information was stored in `/var/lock` and `/var/run`, which are now usually just symbolic links to `/run` or to a subdirectory of `/run`.

/lost+found In certain types of filesystem corruption, fsck (the equivalent of Windows checkdsk) puts in there orphan files that existed on the disk but did not have a directory entry. I've been using Unix systems for 25 years now, and I've had plenty of power failures while the system was on, and many of them were in the old times without journaling, and yet I believe I've only once seen files in that directory, and they were not useful to me. It's more a legacy directory, and many modern filesystems, such as XFS, don't have it at all. You will not use it, let alone back it up.

/boot This directory contains the stuff essential to boot the system, namely the boot loader and the Linux kernel. The installer creates it and you normally don't need it in backup.

/tmp, /var/tmp `/tmp` is for temporary files; any file you create there will be deleted in the next reboot. If you want to create a temporary file that

will survive reboots, use `/var/tmp`.

/media, /mnt Unlike Windows, where disks and disk-like devices get a letter (C:, D:, E: and so on), in Unix there is a single directory tree. There is only one `/bin`. So, assume you have two disks. How do you access the second disk? The answer is that you "mount" it on a point of the directory tree. For example, a relatively common setup for multiuser systems is for the second disk to contain the `/home` directory with the user data, and for the first disk to contain all the rest. In that case, after the system boots, it will mount the second disk at `/home`, so if you `ls /home` you will see the contents of the second disk (if the first disk also has files inside the `/home` directory, these will become hidden and inaccessible after the second disk is mounted).

The `/media` directory is used mostly in desktop systems. If you plugin a USB memory stick or a CDROM, it is usually mounted in a subdirectory of `/media`. The `/mnt` directory exists only as a facility for the administrator, whenever there is a need to temporarily mount another disk. These two directories are rarely used in small virtual servers.

/var/cache As its name implies, this directory is for cached data. Anything in it can be recreated. Its purpose is to speed things up, for example by keeping local copies of things whose canonical position is somewhere in the network. It can be quite large and it would be a waste of storage to back it up.

/var/swap, /var/swapfile, /var/swap.img These are nonstandard files that some administrators use for swap space (swap space is what Windows incorrectly calls "virtual memory"). Swap space is normally placed on dedicated disk partitions. If your system doesn't have such files, so much the better, but keep these files excluded because in the future you or another administrator might create them.

/var/crash If the system crashes the kernel may dump some debugging information in there.

/var/lib/mysql, /var/lib/postgresql We won't directly backup your

databases. Section "Backing up databases" explains why and how.

One more directory that is giving me headaches is `/var/lib/lxcfs`. Like `/proc`, it creates error messages when you try to walk through. It is related to LXC, a virtual machine technology, which seems to be installed on Ubuntu by default (at least in DigitalOcean). I think it could be a bad idea to exclude it, in case you start using LXC in the future and forget it's not being backed up. I just remove LXC with `apt purge lxc-common lxcfs` and I'm done, as this also removes the directory.

9.8 Additional directories for excluding or including

Your backup system will work well if you exclude only the directories I already mentioned. In this section I explain what the other directories are and I discuss whether and under what circumstances they should be excluded.

/bin, /lib, /sbin `/bin` and `/sbin` contain executable programs. For example, if you list the contents of `/bin`, you will find that `ls` itself is among the files listed. The files in `/bin` and `/sbin` are roughly the equivalent of the .EXE files in `C:\Windows\System32`. The difference between `/bin` and `/sbin` is that programs in `/bin` are intended to be run by all users, whereas the ones in `/sbin` are for administrators only. For example, all users are expected to want to list their files with `ls`, but only administrators are expected to partition disks with `fdisk`, which is why `fdisk` is `/sbin/fdisk`.

`/lib` contains shared libraries (the equivalent of Windows Dynamic Link Libraries). The files in `/lib` are roughly the equivalent of the .DLL files in `C:\Windows\System32`. One difference is that in `C:\Windows\System32` you may also find DLLs installed by third-party software; in `/lib`, however, there are only shared libraries essential for the operation of the system.

There may also be other `/lib` directories, such as `/lib32` or `/lib64`. These also contain essential shared libraries. On my 64-bit systems the libraries are actually in `/lib`, but there also exists `/lib64`, which only contains a symbolic link to a library in `/lib`. On other systems `/lib` may be a symbolic to either `/lib32` or `/lib64`. In any case, the system manages all these directories itself and we usually don't need to care.

/etc As we have already said in *Users and directories*, `/etc` contains configuration files.

/home, /root `/home` is where user files are stored. It's the equivalent of Windows' `C:\Users` (formerly `C:\Documents and Settings`). However, the root user doesn't have a directory under `/home`; instead, the home directory for the root user is `/root`. Since the root user is only meant to do administrative work on a system and not to use it and create files like a normal user, the `/root` directory is often essentially empty and unused. However, if you want to create some files it's an appropriate place.

Very often in servers `/home` is also empty, since there are no real users (people), but this actually depends on how the administrator decides to setup the system. For example, some people may create a django user with a `/home/django` directory and install their django project in there. In this book we have created a user, but we have been using different directories for the Django project, as explained in previous chapters.

/usr, /opt, /srv `/usr` has nothing to do with users, and its name is a historical accident. It's the closest thing there is to Windows' `C:\Program Files`. Everything in `/usr` is in subdirectories.

`/usr/bin`, `/usr/lib`, and `/usr/sbin` are much like `/bin`, `/lib` and `/sbin`. The difference is that the latter contain the most essential utilities and libraries of the operating system, whereas the ones under `/usr` contain stuff from add-on packages and the less important utilities. Nowadays the distinction is not important, and I think that lately some systems are starting to make `/bin` a link to `/usr/bin` and so on. It used to be important when the disks were small and the whole of `/usr` was

on another disk that was being mounted later in the boot process.

I'm not going to bother you with more details about the `/usr` subdirectories, except `/usr/local`. Everything installed in `/usr`, except `/usr/local`, is managed by the Debian/Ubuntu package manager. For example, `apt` will install programs in `/usr`, but will not touch `/usr/local`. Likewise, while you can modify stuff inside `/usr/local`, you should not touch any other place under `/usr`, because this is supposed to be managed only by the system's package manager. The tools you use respect that; for example, if you install a Python module system-wide with `pip`, it will install it somewhere under `/usr/local/lib` and/or `/usr/local/share`. `/usr/local` has more or less the same subdirectories as `/usr`, and the difference is that only you (or your tools) write to `/usr/local`, and only the system package manager writes to the rest of `/usr`.

Programs not installed by the system package manager should go either to `/usr/local`, or to `/opt`, or to `/srv`. Here is the theory:

- If the program replaces a system program, use `/usr/local`. For example, a few pages ago I explained how we can install duplicity on Debian 8. The installation procedure I specified will by default put it in `/usr/local`.
- If the program, its configuration and its data are to be installed in a single directory, it should be a subdirectory of `/srv`.
- If the program directories are going to be cleanly separated into executables, configuration, and data, the program should go to `/opt` (and the configuration to `/etc/opt`, and the data to `/var/opt`). This is what we have been doing with our Django project throughout this book.

This subtle distinction is not always followed in practice by all people, so you should be careful with your assumptions.

On carefully setup systems, you don't need to backup `/bin`, `/lib`, `/sbin`, `/usr` and `/opt`, because you can recreate them by re-installing the programs.

This is true particularly if you are setting up your servers using some kind of automation system. I use Ansible. If a server explodes, I create another one, I press a button, and Ansible sets up the server in a few minutes, installing and configuring all necessary software. I only need to restore the data. In theory (and in practice) I don't need `/etc` either, but I never exclude it from backup, it's only about 10 MB anyway. So, in theory, the only directories you need to backup are `/var`, `/srv`, `/root` and `/home`.

> **Warning:** Specify what you want to exclude, not what you want to backup
>
> If you decide that only a few directories are worth backing up, it may be tempting to tell the system "backup directories X, Y and Z" instead of telling it "backup the root directory and exclude A, B, C, D, E, F, G, H, I and J". Don't do it. In the future, you or another administrator will create a directory such as `/data` and put critical stuff in there, and everyone will forget that it is not being backed up. Always backup the root file system and specify what you want to exclude, not what you want to include.

If you aren't using automation (and this could fill another book on its own), it would be better to not exclude `/opt` from backup, because it will make it harder to recover. It's very unlikely `/bin`, `/lib` and `/sbin` will be useful when restoring, but they're not much disk space anyway. The only real question is whether to backup `/usr`, which can be perhaps 1 GB. At $0.58 per year it's not much, but it might also make backup and restore slower.

Is your head spinning? Here's the bottom line: use the exclude list provided in the previous section, and if you feel confident also exclude `/bin`, `/lib`, `/sbin` and `/usr`. If your Django project's files in `/opt` consume much space, and you believe you can re-clone them fast and setup the virtualenv fast (as described in *Users and directories*), you can also exclude `/opt`.

Whatever you decide, you might make an error. You might accidentally exclude something crucial. This is true even if you don't exclude anything at all. For example, if you keep encrypted backups, you might think you

are saving everything but you might be forgetting to store the decryption password somewhere.

The only way to be reasonably certain you are not screwing up is to test your recovery as I explain later.

Tip: Check the disk space

Two commands you will find useful are `df` and `du`.

```
df -h
```

This shows the disk space usage for all the file systems. You are normally only interested for the file system that is mounted on "/", which is something like `/dev/sda1` or `/dev/vda1`. This is your main disk.

```
cd /
du -sh *
```

This will calculate and display the disk space that is occupied by each directory. It will throw some error messages, which can be ignored.

A useful variation is this:

```
du -sh * | sort -h
```

This means "take the standard output of `du -sh *` and use it as standard input to `sort -h`". The standard output does not include the error messages (these go to the standard error). `sort` is a program that sorts its input; with the `-h` option, it sorts human readable byte counts such as "15M" and "1.1G".

If the output of `du` is longer than your terminal, another useful idiom is this:

```
du -sh * | sort -h | less
```

This will take the standard output of `sort` and give it as input to `less`. `less` is a program that only shows only one screenful of information at a

time. If you get accustomed to it you'll find it's much more convenient than using the scrollbar of your terminal. You can use j and k (or the arrow keys) to go down and up, space and b (or Page Down/Up) for the next and previous screenful, G and g to go to the end and beginning, and q to exit. You can also search with a slash, and repeat a search forwards and backwards with n and N.

9.9 Backing up databases

Databases cannot usually be backed up just by copying their data files. For small databases, copying can take a few seconds or a few minutes. During this time, the files could be changing. As a result, when you restore the files, the database might not be internally consistent. Even if you ensure that no-one is writing to the database, or even that there are no connections, you can still not copy the files, because the RDBMS may be caching information and flushing it whenever it likes. To backup by copying data files you need to shutdown the RDBMS, which means downtime.

The problem of internal consistency is also present with SQLite. Copying the database file can take some time, and if the database is being written to during that time, the file will be internally inconsistent, that is, corrupt.

Backing up large databases involves complicated strategies, such as those described in Chapter 25 of the PostgreSQL 9.6 manual. Here we are going to follow the simplest strategy which is to dump all the database to a plain text file. Database dumps are guaranteed to be internally consistent. SQLite may lock the database during the dump, meaning writing to it will have to wait, but the time you need to wait for small databases is very little.

For **PostgreSQL**, create file `/etc/duply/main/pre`, with the following contents:

```
#!/bin/bash
su postgres -c 'pg_dumpall --file=/var/backups/postgresql.dump'
```

For **SQLite**, the contents of `/etc/duply/main/pre` should be:

```
#!/bin/bash
echo '.dump' | \
   sqlite3 /var/opt/$DJANGO_PROJECT/$DJANGO_PROJECT.db \
       >/var/backups/sqlite-$DJANGO_PROJECT.dump
```

Better let's make `/etc/duply/main/pre` executable:

```
chmod 755 /etc/duply/main/pre
```

The file is actually a **shell script**. In their simplest form, shell scripts are just commands one after the other (much like Windows `.bat` files). However, Unix shells like bash are complete programming languages (in fact duply itself is written in bash). We won't do any complicated shell programming here, but if, for some reason, you have both PostgreSQL and SQLite on a server, you can join the two above scripts like this:

```
#!/bin/bash
su postgres -c 'pg_dumpall --file=/var/backups/postgresql.dump'
echo '.dump' | \
   sqlite3 /var/opt/$DJANGO_PROJECT/$DJANGO_PROJECT.db \
       >/var/backups/sqlite-$DJANGO_PROJECT.dump
```

Likewise, if you have many SQLite databases, you need to add a dump command for each one in the file (this is not necessary for PostgreSQL, as `pg_dumpall` will dump all databases of the cluster).

Duply will execute `/etc/duply/main/pre` before proceeding to copy the files. (It will also execute `/etc/duply/main/post`, if it exists, after copying, but we don't need to do anything like that; with different backup schemes `pre` could, for example, shutdown the database and `post` could start it again.)

If you don't understand the `pre` file for SQLite, here is the explanation: to

dump a SQLite database, you connect to it with `sqlite3 dbname` and then execute the SQLite `.dump` command. The `sqlite3` program reads commands from the standard input and writes dumps to the standard output. The standard input is normally your keyboard; but by telling it `echo '.dump' | sqlite3 ...` we give it the string ".dump", followed by newline, as standard input (the `echo` command just displays stuff and follows it with a newline; for example, try `echo 'hello, world'`). The vertical line, as I explained in the previous section (see *Check the disk space*) sends the output of one command as input to another command. Finally, the ">" is the **redirection** symbol, it redirects the standard output of the `sqlite3` program, which would otherwise be displayed on the terminal, to a file.

Tip: Compressing database dumps

Database dumps are plain text files. If compressed, they can easily become five times smaller. However, compressing them might make incremental backups larger and slower. The reason is that in incremental backups duplicity saves only what has changed since the previous backup. It might be easier for duplicity to detect changes in a plain text file than in a compressed file, and the result could be to backup the entire compressed file each time. Since duplicity compresses backups anyway, storing the dump file uncompressed will never result in larger backups.

The only downside of storing the dump file uncompressed is that it takes up more disk space in the server. This is rarely a problem.

Tip: Excluding SQLite

Technically, since you are dumping the database, you should be excluding `/var/opt/$DJANGO_PROJECT/$DJANGO_PROJECT.db`, from the backup; however if the database file is only a few hundreds of kilobytes the savings aren't worth the trouble of adding it to your `exclude` file.

9.10 Running scheduled backups

Create file /etc/cron.daily/duply with the following contents:

```
#!/bin/bash
duply main purge --force >/tmp/duply.out
duply main purgeIncr --force >>/tmp/duply.out
duply main backup >>/tmp/duply.out
```

Make the file executable:

```
chmod 755 /etc/cron.daily/duply
```

We saw about cron in *Clearing sessions*. In the /etc/cron.daily/duply script, the first command, purge, will delete full backups that are older than MAX_AGE. The second command, purgeIncr, will delete incremental backups that build on full backups that are older than MAX_FULLS_WITH_INCRS. Finally, the third command, backup, will perform an incremental backup, unless a full backup is due. A full backup is due if you have never backed up in the past, or if the latest full backup was done more than MAX_FULLBKP_AGE ago.

Duply displays a lot of information even when everything's working fine, which would result in cron to email the administrator. We only want to be emailed in case of error, so we redirect duply's output to a file, /tmp/duply.out. We only redirect its standard output, not its standard error, which means that error (and warning) messages will still be caught by cron and emailed. Note, however, that /tmp/duply.out is not a complete log file, because it only contains the standard output, not the standard error. It might have been better to include both output and error in /tmp/duply.out, and in addtion display the standard error, so that cron can catch it; however, this requires more advanced shell scripting techniques and it's more trouble than it's worth.

The redirection for the first command, >/tmp/duply.out, overwrites /tmp/duply.out if it already exists. The redirection for the next two commands, >>/tmp/duply.out, appends to the file.

Warning: You must use a local mail server

The emails of cron cannot be sent unless a mail server is installed locally on the server. See *Using a local mail server* to setup one. Don't omit it, otherwise you won't know when your system has a problem and cannot backup itself.

9.11 Chapter summary

- Keep some offline backups and regularly test recovery (the next chapter deals with these).
- Calculate storage costs.
- Create a bucket in your backup storage. A single bucket for all your deployments is probably enough. You can name it `$NICK-backup`.
- Install duply, create directory `/etc/duply/main`, and chmod it to 700.
- Create configuration file `/etc/duply/main/conf` with these contents:

```
GPG_KEY=disabled

SOURCE=/
TARGET=b2://$ACC_ID:$APP_KEY@$NICK-backup/$SERVER_NAME/

MAX_AGE=2Y
MAX_FULLS_WITH_INCRS=2
MAX_FULLBKP_AGE=3M
DUPL_PARAMS="$DUPL_PARAMS --full-if-older-than $MAX_
 ↪FULLBKP_AGE "

VERBOSITY=warning
ARCH_DIR=/var/cache/duplicity/duply_main/
```

- Create file `/etc/duply/main/exclude` with the following contents:

```
- /dev
- /proc
- /sys
- /run
- /var/lock
- /var/run
- /lost+found
- /boot
- /tmp
- /var/tmp
- /media
- /mnt
- /var/cache
- /var/crash
- /var/swap
- /var/swapfile
- /var/swap.img
- /var/lib/mysql
- /var/lib/postgresql
```

  If you feel like it, also exclude `/bin`, `/lib`, `/sbin` and `/usr`, maybe also `/opt`.

- Create file `/etc/duplicity/main/pre` with contents similar to the following (delete the PostgreSQL or SQLite part as needed, or add more SQLite commands if you have many SQLite databases):

```
#!/bin/bash
su postgres -c 'pg_dumpall --file=/var/backups/postgresql.
 ↪dump'
echo '.dump' | \
   sqlite3 /var/opt/$DJANGO_PROJECT/$DJANGO_PROJECT.db \
       >/var/backups/sqlite-$DJANGO_PROJECT.dump
```

  Chmod the file to 755.

- Create file `/etc/cron.daily/duply` with the following contents:

```
#!/bin/bash
duply main purge --force >/tmp/duply.out
duply main purgeIncr --force >>/tmp/duply.out
duply main backup >>/tmp/duply.out
```

  Chmod the file to 755.

- Make sure you have a local mail server installed.

CHAPTER

TEN

RECOVERY PART 2

10.1 Restoring a file or directory

You made some changes to `/etc/opt/$DJANGO_PROJECT/settings.py`, changed your mind, and you want it back? No problem:

```
duply main fetch etc/opt/$DJANGO_PROJECT/settings.py \
   /tmp/restored_settings.py
```

This will fetch the most recent version of the file from backup and will put it in `/tmp/restored_settings.py`. Note that when you specify the source file there is no leading slash.

You can also fetch previous versions of the file:

```
# Fetch it as it was 4 days ago
duply main fetch etc/opt/$DJANGO_PROJECT/settings.py \
   /tmp/restored_settings.py 4D

# Fetch it as it was on 4 January 2017
duply main fetch etc/opt/$DJANGO_PROJECT/settings.py \
   /tmp/restored_settings.py 2017-01-04
```

Here is how to restore all the backup into `/tmp/restored_files`:

```
duply main restore /tmp/restored_files
```

As before, you can append age specifiers such as `4D` or `2017-01-04` to the command. Note that restoring a large backup can incur charges by your backup storage provider.

You should probably never restore files directly to their original location. Instead, restore into `/tmp` or `/var/tmp` and move or copy them.

10.2 Restoring SQLite

Restoring SQLite is very simple. Assuming the dump file is in `/tmp/restored_files/var/backups/sqlite-$DJANGO_PROJECT.dump`, you should be able to recreate your database file thus:

```
sqlite3 /tmp/$DJANGO_PROJECT.db \
    </tmp/restored_files/var/backups/sqlite-$DJANGO_PROJECT.dump
```

This will create `/tmp/$DJANGO_PROJECT.db` and it will execute the commands in the dump file. You can then move the file to its normal position, such as `/var/opt/$DJANGO_PROJECT/$DJANGO_PROJECT.db`. You probably need to `chown` it to $DJANGO_USER.

10.3 Restoring PostgreSQL

How you will restore PostgreSQL depends on what exactly you want to restore and what the current state of your cluster is. For a moment, let's assume this:

1. You have just installed PostgreSQL with `apt install postgresql` and it has created a brand new cluster that only contains the databases `postgres`, `template0` and `template1`.
2. You want to restore all your databases.

Assuming `/tmp/restored_files/var/backups/postgresql.dump` is the dump file, you can do it this way:

```
cd /tmp/restored_files/var/backups
su postgres -c 'psql -f postgresql.dump postgres' >/dev/null
```

`psql` shows a lot of output, which we don't need. We redirect the output to `/dev/null`, which in Unix-like systems is a black hole; it is a device file that merely discards everything written to it. We discard only the standard output, not the standard error, because we want to see error messages. If everything goes well, it should show only one error message:

> ERROR: role "postgres" already exists

The file written to by `pg_dumpall` contains SQL commands that can be used to recreate all databases. In the beginning of the file there are commands that first create the users. One of these users is `postgres`, but this already exists in your new cluster, therefore the error message. (The dump file also includes commands to create the databases, but `pg_dumpall` is smart enough to not include database creation commands for template0, template1, and postgres.)

Hint: Playing with redirections

You might want to redirect the standard error as well as the standard output. You can do it like this:

```
su postgres -c 'psql -f postgresql.dump postgres' \
   >/tmp/psql.out 2>/tmp/psql.err
```

This actually means "redirect file descriptor 1 to /tmp/psql.out and file descriptor 2 to /tmp/psql.err". Instead of `>file` you can write `1>file`, but

1 is the default and custom has it to omit it almost always. File descriptor 1 is always standard output, and 2 is always standard error. There are several use cases for redirecting the standard error, and one of them is if you want to keep a record of the error messages so that you can examine them later.

One problem is that `psql` actually throws error messages interspersed with standard output messages, and if you separate output from error you might not know at which stage the error occurred. If you want to log the error messages in the same file and in the correct position in relation to the output messages, you can do this:

```
su postgres -c 'psql -f postgresql.dump postgres' \
   >/tmp/psql.out 2>&1
```

The `2 > &1` means "redirect the standard error to the same place where you're putting the standard output".

However, this will not always work as you expect because the standard output is buffered whereas the standard error is unbuffered; so sometimes error messages can appear in the file **before** output that was supposed to be printed before the error.

If something goes wrong and you want to start over, here is how, but **be careful not to type these in the wrong window** (you could delete a production cluster in another server):

```
service postgresql stop
pg_dropcluster 9.5 main
pg_createcluster 9.5 main
service postgresql start
```

The second command will remove the "main" cluster of PostgreSQL version 9.5 (replace that with your actual PostgreSQL version). The third command will initialize a brand new cluster.

10.4 Restoring an entire system

A few sections ago we saw how to restore all backed up files in a temporary directory such as `/tmp/restored_files`. If your server (the "backed up server") has exploded, you might be tempted to setup a new server (the "restored server") and then just restore all the backup directly in the root directory instead of a temporary directory. This won't work correctly, however. For example, if you restore all of `/var/lib`, you will overwrite `/var/lib/apt` and `/var/lib/dpkg`, where the system keeps track of what packages it has installed, so it will think it has installed all the packages that had been installed in the backed up server, and the system will essentially be broken. Or if you restore `/etc/network` you might overwrite the restored system's network configuration with the network configuration of the backed up server. So you can't do this; you need restore the backup in `/tmp/restored_files` and then selectively move or copy stuff from there to its normal place.

Below I present a complete recovery plan that you can use whenever your system needs recovery. It should be applicable in its entirety only when you need a complete recovery; however, if you need a partial recovery you can still follow it and omit some parts as you go. **I assume the backed up system only had Django apps deployed in the way I have described in the rest of this book.** If you have something else installed, or if you have deployed in a different way (e.g. in different directories), you **must** modify the plan with one of your own.

You must also make sure that you have access to the recovery plan even if the server goes down; that is, don't store the recovery plan on a server that is among those that may need to be recovered.

> **Hint:** The rm command
>
> In various places in the following recovery plan, I tell you to use the `rm` command, which is the Unix command that removes files. With the `-r` option it recursively removes directories, and `-f` means "ask no questions".

The following will delete the nginx configuration, asking no questions:

```
rm -rf /etc/nginx
```

`rm` accepts many arguments, so `rm -rf /etc/nginx /etc/apache2` will delete both directories. Accidentally inserting a space, as in `rm -rf / etc/nginx`, will delete mostly all your system.

AAA.

1. Notify management, or the customer, or whoever is affected and needs to be informed.
2. Take notes. In particular, mark on this recovery plan anything that needs improvement.
3. Create a new server and add your ssh key.
4. Change the DNS so that $DOMAIN, www.$DOMAIN, and any other needed name points to the IP address of the new server (see *Adding records to your domain*).
5. Create a user and group for your Django project (see *Creating a user and group*).
6. Install packages:

   ```
   apt install python python3 \
      python-virtualenv python3-virtualenv \
      postgresql python-psycopg2 python3-psycopg2 \
      sqlite3 dma nginx-light duply
   ```

   (Ignore questions on how to setup dma, we will restore its configuration from the backup later.)

   If you use Apache, install `apache2` instead of `nginx-light`. The actual list of packages you need might be different (but you can also find this out while restoring).

7. Check duplicity version with `duplicity --version`; if earlier than 0.7.6 and your backups are in Backblaze B2, install a more recent version of duplicity as explained in *Installing duplicity in Debian*.
8. Create the duply configuration directory and file as explained in *Setting up duplicity and duply* (you don't need to create any files beside `conf`, you don't need `exclude` or `pre`).
9. Restore the backup in `/var/tmp/restored_files`:

```
duply main restore /var/tmp/restored_files
```

10. Restore the `/opt`, `/var/opt` and `/etc/opt` directories:

```
cd /var/tmp/restored_files
cp -a var/opt/* /var/opt/
cp -a etc/opt/* /etc/opt/
cp -a opt/* /opt/
```

    (If you have excluded `/opt` from backup, clone/copy your Django project in `/opt` and create the virtualenv as described in *The program files*.)

11. Create the log directory as explained in *The log directory*.
12. Restore your nginx configuration:

```
service nginx stop
rm -r /etc/nginx
cp -a /var/tmp/restored_files/etc/nginx /etc
service nginx start
```

    If you use Apache, restore your Apache configuration instead:

```
service apache2 stop
rm -r /etc/apache2
cp -a /var/tmp/restored_files/etc/apache2 /etc/
service apache2 start
```

13. Create your static files directory and run `collectstatic` as explained in *Static and media files*.
14. Restore the systemd service file for your Django project and enable the service:

```
cd /var/tmp/restored_files
cp etc/systemd/system/$DJANGO_PROJECT.service \
    /etc/systemd/system/
systemctl enable $DJANGO_PROJECT
```

15. Restore the configuration for the DragonFly Mail Agent:

```
rm -r /etc/dma
cp -a /var/tmp/restored_files/etc/dma /etc/
```

16. Create the cache directory as described in *Caching*.
17. Restore the databases as explained in *Restoring SQLite* and *Restoring PostgreSQL*.
18. Restore the duply configuration:

```
rm -r /etc/duply
cp -a /var/tmp/restored/files/etc/duply /etc/
```

19. Restore the `duply` cron job:

```
cp /var/tmp/restored/etc/cron.daily/duply /etc/cron.daily/
```

    (You may want to list `/var/tmp/restored/etc/cron.daily` and `/etc/cron.daily` to see if there is any other cronjob that needs restoring.)

20. Start the Django project and verify it works:

```
service $DJANGO_PROJECT start
```

21. Restart the system and verify it works:

```
shutdown -r now
```

The system might work perfectly without restart; the reason we restart it is to verify that if the server restarts, all services will startup properly.

After you've finished, update your recovery plan with the notes you took.

10.5 Recovery testing

In the previous chapter I said several times that you must test your recovery. Your recovery testing plan depends on the extent to which downtime is an issue.

If downtime is not an issue, that is, you can find a date and time in which the system is not being used, the simplest way to test the recovery is to shutdown the server, pretend it has been entirely deleted, and follow the recovery plan in the previous section to bring the system up on a new server. Keep the old server off for a week or a month or until you feel confident it really has no useful information, then delete it.

If you can't have much downtime, maybe there are times when the system is not being written to. Many web apps are like this; you want them to always be readable by the visitors, but maybe they are not being updated off hours. In that case, notify management or the customer about what you are going to do, pick up an appropriate time, and test the recovery with the following procedure:

1. In the DNS, verify that the TTL of $DOMAIN, www.$DOMAIN, and any other necessary record is no more than 300 seconds or 5 minutes (see *Adding records to your domain*).
2. Follow the recovery plan of the previous section to bring up the system on a new server, **but omit the step about changing the DNS**.

(Hint: you can *edit your own hosts file* while checking if the new system works.)

3. After the system works and you've fixed all problems, change the DNS so that $DOMAIN, www.$DOMAIN, and any other needed name points to the IP address of the new server (see *Adding records to your domain*).
4. Wait for five minutes, then shut down the old server.

You could have zero downtime by only following the first two steps instead of all four, and after you are satisfied discard the *new* server instead of the old one. However, you can't really be certain you haven't left something out if you don't use the new server operationally. So while following half the testing plan can be a good idea as a preliminary test in order to get an idea of how much time will be needed by the actual test, staying there and not doing the actual test is a bad idea.

If you think you can't afford any downtime at all, you are doing something wrong. You *will* have downtime when you accidentally delete a database, when there is a hardware or network error, and in many other cases. Pretending you won't is a bad idea. If you really can't afford downtime, you should setup high availability (which is a lot of work and can fill in several books by itself). If you don't, it means that the business *can* afford a little downtime once in a while, so having a little scheduled downtime once a year shouldn't be a big deal.

In fact, I think that, in theory at least, recovery should be tested during business hours, possibly without notifying the business in advance (except to get permission to do it, but not to arrange a specific time). Recovery isn't merely a system administrator's issue, and an additional recovery plan for management might need to be created, that describes how the business will handle the situation (what to tell the customers, what the employees should do, and so on). Recovery with downtime during business hours can be a good exercise for the whole business, not just for the administrator.

10.6 Copying offline

Briefly, here is how to copy the server's data to your local machine:

```
awk '{ print $2 }' /etc/duply/main/exclude >/tmp/exclude
tar czf - --exclude-from=/tmp/exclude / | \
    split --bytes=200M - \
        /tmp/`hostname`-`date --iso-8601`.tar.gz.
```

This will need some explanation, of course, but it will create one or more files with filenames similar to the following:

```
/tmp/myserver-2017-01-22.tar.gz.aa
/tmp/myserver-2017-01-22.tar.gz.ab
/tmp/myserver-2017-01-22.tar.gz.ac
```

We will talk about downloading them later on. Now let's examine what we did. We will check the last command (i.e. the `tar` and `split`) first.

We've seen the `tar` command earlier, in *Installing duplicity in Debian*. The "c" in "czf" means we will create an archive; the "z" means the archive will be compressed; the "f" followed by a file name specifies the name of the archive; "f" followed by a hyphen means the archive will be created in the standard output. The last argument to the `tar` command specifies which directory should be put in the archive; in our case it's a mere slash, which means the root directory. The `--exclude-from=/tmp/exclude` option means that files and directories specified in the `/tmp/exclude` file should not be included in the archive.

This would create an archive with all the files we need, but it might be too large. If your external disk is formatted in FAT32, it might not be able to hold files larger than 2 GB. So we take the data thrown at the standard output and we split it in manageable chunks of 200 MB each. This is what the `split`

command does. The hyphen in `split` means "split the standard input". The last argument to `split` is the file prefix; the files `split` creates are named `PREFIXaa`, `PREFIXab`, and so on.

The backticks in the specified prefix are a neat shell trick: the shell executes the command within the backticks, takes the command's standard output, and inserts it in the command line. So the shell will first execute `hostname` and `date --iso-8601`, it will then create the command line for `split` that contains among other things the output of these commands, and then it will execute `split` giving it the calculated command line. We have chosen a prefix that ends in `.tar.gz`, because that is what compressed tar files end in. If you concatenate these files into a single file ending in `.tar.gz`, that will be the compressed tar file. We will see how to concatenate them two sections ahead.

Finally, let's explain the first command, which creates `/tmp/exclude`. We want to exclude the same directories as those specified in `/etc/duply/main/exclude`. However, the syntax used by duplicity is different from the syntax used by `tar`. Duplicity needs the pathnames to be preceded by a minus sign and a space, whereas `tar` just wants them listed. So the first command merely strips the minus sign. `awk` is actually a whole programming language, but you don't need to learn it (I don't know it either). The `{ print $2 }` means "print the second item of each line". While `awk` is the canonical way of doing this in Unix-like systems, you could do it with Python if you prefer, but it's much harder:

```
python -c "import sys;\
    print('\n'.join([x.split()[1] for x in sys.stdin]))" \
    </etc/duply/main/exclude >/tmp/exclude
```

Now let's **download the archive**. That's easy using `scp` (on Unix-like systems) or `pscp` (on Windows). Assuming the external disk is plugged in and available as $EXTERNAL_DISK (i.e. something like `/media/user/DISK` on GNU/Linux, and something like `E:\` on Windows), you can put it directly in there like this:

```
scp root@$SERVER_IP_ADDRESS:/tmp/*.tar.gz.* $EXTERNAL_DISK
```

In Windows, use `pscp` instead of `scp`. You can also use graphical tools, however command-line tools can often be more convenient.

In Unix-like systems, a better command is `rsync`:

```
rsync root@$SERVER_IP_ADDRESS:/tmp/*.tar.gz.* $EXTERNAL_DISK
```

If for some reason the transfer is interrupted and you restart it, `rsync` will only transfer the parts of the files that have not yet been transferred. `rsync` must be installed both on the server and locally for this to work. You may have success with Windows rsync programs such as DeltaCopy.

One problem with the above scheme is that we temporarily store the split tar file on the server, and the server might not have enough disk space for that. In that case, if you run a Unix-like system locally, this might work:

```
ssh root@$SERVER_IP_ADDRESS \
    "awk '{ print \$2 }' /etc/duply/main/exclude
       >/tmp/exclude; \
     tar czf - --exclude-from=/tmp/exclude  /" | \
  split --bytes=200M - \
    $EXTERNAL_DISK/$SERVER_NAME-`date --iso-8601`.tar.gz.
```

The `ssh` command will login to the remote server and execute the commands `awk` and `tar`, and it will capture their standard output (i.e. `tar`'s standard output, because `awk`'s is redirected) and it will throw it in its own standard output.

The trickiest part of this `ssh` command is that, in the `awk`, we have escaped the dollar sign with a backslash. `awk` is a programming language, and `{ print $2 }` is an `awk` program. `awk` must literally receive the string `{ print $2 }` as its program. When we give a local shell the command `awk '{ print $2 }'`, the shell leaves the `{ print $2 }` as it is, because it is enclosed in single quotes. If, instead, we used double quotes, we would use `awk "{ print`

`\$2 }"`, otherwise, if we simply used `$2`, the shell would try to expand it to whatever `$2` means (see *Bash syntax*). Now the string given to `ssh` is a double-quoted string. The *local* shell gets that string and performs expansions and runs `ssh` after it has done these expansions; and `ssh` gets the resulting string, executes a shell remotely, and gives it that string. You can understand the rest of the story with a bit of thinking.

If you aren't running a Unix-like system locally, something else you can do is use another Debian/Ubuntu server that you have on the network and does have the disk space. You can also temporarily create one at Digital Ocean just for the job. After running the above command to create the backup and store it in the temporary server, you can then copy it to your local machine and external disk.

You may have noticed we did not backup the databases. I assume that your normal backup script does this every day, and it stores the saved databases in `/var/backups`. You need to be careful, however, to not run the `tar` command at the same time cron and duply run `/etc/duply/main/pre`, otherwise you might be copying them at exactly the time they are being overwritten.

10.7 Storing and rotating external disks

In the previous chapter I told you you need two external disks. Store one of them at your office and the other elsewhere—at your home, at your boss's home, at a bank vault, at a backup storage company, or at your customer's office or home (however don't give your customer a disk that also contains data of other customers of yours). Whatever place you chose, I will be calling it "off site". So you will be keeping one disk off site and one on site. Whenever you want to perform an offline backup (say once per month), connect the disk you have on site, delete all the files it contains, and perform the procedure described in the previous section to backup your servers on it. After that, physically label it with the date (overwriting or removing the previous label), and move it off site. Bring the other disk on site and let it sit there until the next offline backup.

Why do we use two disks instead of just one? Well, it's quite conceivable that your online data (and online backup) will be severely damaged, and you can perform an offline backup, wiping out the previous one, before realizing the server's severely damanged. In that case, your offline disk will contain damaged data. Or the attacker might wait for you to plug in the backup disk, and then wipe it out and proceed to wipe out the online backup and your servers.

You might object that there is a weakness to this plan because the two disks are at the same location, off site, when you take there the recently used disk and exchange it with the older one. I wouldn't worry too much about this. Offline backups are extra backups anyway, and you hope to never need to use them. While it's possible that someone can get access to all your passwords and delete all your online servers and backups, the probability of this happening at the same time as the physical destruction of your two offline disks at the limited time they are both off site is so low that you should probably worry more about your plane crashing.

With this scheme, you might lose up to one month of data. Normally this is too much, but maybe for the extreme case we are talking about it's OK. Only you can judge that. If you think it's unacceptable, you might perform offline backups more often. If you do them more often than once every two weeks, it would be better to use more external disks.

10.8 Recovering from offline backups

You will probably never need to recover from offline backups, so we won't go into much detail. If a disaster happens and you need to restore from offline, the most important thing you need to care about is the safety of your external disk. Make **absolutely certain** you will only plug it on a safe computer, one that is certainly not compromised by any attacker. Do this very slowly and think about every step. After plugging the external disk in, copy its files to the computer's disk, then unplug the external disk immediately and keep it safe.

Recovery is the same as what's described in *Restoring an entire system*, except for the steps that use duply and duplicity to restore the backup in `/var/tmp/restored_files`. Instead, copy the `.tar.gz.XX` files to the server's `/var/tmp` directory; use `scp` or `pscp` or `rsync` for that (`rsync` is the best if you have it). When you have them all, join them in one piece with the concatenation command, `cat`, then untar them:

```
cd /tmp
cat *.tar.gz.* >backup.tar.gz
mkdir restored_files
cd restored_files
tar xf ../backup.tar.gz
```

If you are low on disk space, you might join the concatenation command with the tar command, like this:

```
cd /tmp
mkdir restored_files
cd restored_files
cat ../*.tar.gz.* | tar xf -
```

10.9 Scheduling manual operations

In the previous chapter, I described stuff that you will eventually set up in such a way that it runs alone. Your servers will be backing up themselves without your knowing anything about it. In contrast, all the procedures I described in this chapter are to be manually executed by a human:

- Restoring part of a system or the whole system
- Recovery testing
- Copying offline
- Recovering from offline backups

Some of these procedures will be triggered by an event, such as losing data. Recovery testing, however, and copying offline, will not be triggered; *you* must take care that they occur. This can be as simple as adding a few recurring entries to your calendar, or as hard as inventing foolproof procedures to be added to the company's operations manual. Whatever you do, you must make sure it works. **If you don't test recovery, it is almost certain it will take too long when you need it, and it is quite likely you will be unable to recover at all.**

10.10 Chapter summary

- Use the provided recovery plan or devise your own.
- Make sure you will have access to the recovery plan (and all required information such as logins and passwords) even if your server stops existing.
- Test your recovery plan once a year or so.
- Backup online as well as to offline disks and store them safely.
- Don't backup to offline disks at the same time as the system is performing its online backup.
- Create an offline backup schedule and a recovery testing schedule and make sure they are being followed.