EXPERT INSIGHT

# Mastering Ubuntu Server

**Gain expertise in the art of deploying, configuring, managing, and troubleshooting Ubuntu Server**

**Third Edition**

**Jay LaCroix**

Packt>

# Mastering Ubuntu Server

*Third Edition*

Gain expertise in the art of deploying, configuring, managing, and troubleshooting Ubuntu Server

**Jay LaCroix**

Packt

BIRMINGHAM - MUMBAI

# Mastering Ubuntu Server

*Third Edition*

**Producer:** Aarthi Kumaraswamy

**Acquisition Editor – Peer Reviews:** Divya Mudaliar

**Content Development Editors:** Edward Doxey, Bhavesh Amin

**Technical Editor:** Aditya Sawant

**Project Editor:** Janice Gonsalves

**Copy Editor**: Safis Editing

**Proofreader:** Safis Editing

**Indexer:** Manju Arasan

**Presentation Designer:** Sandip Tadge

First published: July 2016

Second edition: May 2018

Third Edition: December 2020

Production reference: 1231220

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-80056-464-0

`www.packt.com`

# Contributors

## About the author

**Jay LaCroix** is a passionate technologist and enjoys working on just about any form of technology he can get his hands on. He spends his free time tinkering with network and server hardware, virtualization, Raspberry Pis, writing scripts, and more.

Of all the technology he works with, Jay is most passionate about Linux and its community, and has over 18 years of experience in the industry. Outside of previous editions of this book, Jay has written *Linux Mint Essentials* and *Mastering Linux Network Administration*, also with Packt. In addition, he's created a YouTube channel (LearnLinuxTV) that covers all things Linux, with videos that cover tutorials, distribution reviews, and more.

Jay holds a master's degree in Information Systems Technology Management from Capella University. His career has included roles in system administration and cloud engineering, and currently he leads a team of talented IT professionals that develops cloud solutions for businesses and maintains server and network infrastructure.

Outside of his career, Jay is a Red Belt in Tang Soo Do through **Professional Karate Schools of America** (**PKSA**) and is within a year of achieving his black belt. He also enjoys classic '90s videogames, and has a collection that includes over 800 physical games and more than 30 consoles.

*I want to send a special thank you to my partner, Alan Sutherland, for his love and support.*

*And to our sons, Alan and Jonathan LaCroix, whom I'm very proud of every day; it's amazing to watch them grow.*

*Thank you to Randy Schapel of Mott Community College (Flint MI) for the amazing class over 18 years ago that got me addicted to Linux in the first place and jump-started my career.*

*And to all of the Kyo Sa Nims and Sa Bom Nims of PKSA, including (but certainly not limited to) Kyo Sa Zieger, Kyo Sa Hargett, Master Germain, Master Rasnick, Mr and Mrs Gerstenberger, Master Collins Jr, Master Collins Sr, Master Prosch, and many others. Your leadership has been an inspiration.*

*I also want to thank all of the fans of LearnLinuxTV – this publication absolutely wouldn't have happened without your support.*

# About the reviewers

**Rafael David Tinoco** is an Ubuntu Linux core developer. He currently holds the position of Ubuntu Server engineer at Canonical, the company behind Ubuntu Linux. He is also a member of the Ubuntu Developer Membership Board.

Rafael is a Linux developer and he has worked for the past 20 years in companies such as Linaro (Linux kernel upstream development/validation engineer for ARM and High Performance Computing (HPC) effort Tech Lead), Canonical (Linux Sustaining Engineer and Tech Lead), IBM (Mainframe Lab Engineer and Tech Lead, with a brief residency at the s390x performance team), Red Hat (Solution Architect), Locaweb (Linux/Email DevOps), Sun Microsystems (Consultant, Systems Engineer, HPC Engineer, and Solaris Ambassador).

Rafael has taken the roles of his career from field engineering positions—in constant contact with customers and critical mission environments—into pure engineering ones, focused in Linux internals, userland, kernel and general OS, and performance debugging and bug solving.

*I would like to thank my wife, Patricia, without whose support I wouldn't be here and would not have produced so much in life – including this book review – my kids, Matheus and Giovanna, for teaching me so much every day, and Canonical for allowing me to participate in this book review. Thanks also to Packt, for inviting me on such a great experience, and to Jay LaCroix for making this book an incredible journey for those who want to learn about Ubuntu Linux as a server.*

**Dawn Bott's** IT career began in a 911 dispatch center as the Dispatch Coordinator. Some of her duties included updating all computers in the police cruisers, providing desktop support for all machines in the center, and resolving any other IT issues that came up in day-to-day operations. When the 911 dispatch center was taken over by the county, she decided to continue her career in the IT world.

She took a job at a global technology solutions provider that supplied services to the automotive industry. She began on the helpdesk, working with dealerships to troubleshoot diagnostic tools and network issues as a level 1 technician, before quickly moving to team lead.

While working on the helpdesk, she went back to school and earned two associate degrees in Technological Sciences and Computer Support Engineering technology, gaining a Cisco Certificate of Achievement.

Once her degree programs were complete, she was promoted to Technical Services System Administrator. She also began assisting the cloud team and working with AWS, and was eventually transferred to the cloud team full time.

After 6 years she moved to Go2Group as a Cloud Architect, where she was reunited with co-workers from a previous employer. Go2Group has since merged with Adaptavist and she is still currently with the company. This is the first book that Dawn has worked on, and it was a great experience.

*I want to thank Jay LaCroix for writing this book and for allowing me the pleasure of assisting with the editing process, my husband, Harry, for his patience during this process, and Packt Publishing for all their support.*

# Table of Contents

# Preface

Ubuntu is an exciting platform. You can literally find it everywhere—desktops, laptops, phones, and especially servers. The server edition enables administrators to create efficient, flexible, and highly available servers that empower organizations with the power of open source. As Ubuntu administrators, we're in good company—according to W3Techs, Ubuntu is the most widely deployed distribution on the web with regard to Linux. With the release of Ubuntu 20.04, this platform becomes even more exciting!

In this book, we will dive right into Ubuntu Server, and you will learn all the concepts needed to manage your servers and configure them to perform all kinds of neat tasks, such as serving web pages, managing virtual machines, running containers, automating configuration, sharing files with other users, and even running Ubuntu in the cloud.

We'll start our journey with the first chapter, where we'll walk through the installation of Ubuntu Server 20.04, which will serve as a foundation for the rest of the book. As we proceed through our journey, we'll look at managing users, connecting to networks, and controlling processes. Later, we'll implement important technologies, such as DHCP, DNS, Apache, MariaDB, and more. We'll even set up our own Nextcloud server along the way.

Finally, the end of the book covers various things we can do to troubleshoot issues, as well as preventing and recovering from disasters.

## Who this book is for

This book is intended for readers with intermediate or advanced-beginner Linux skills, who would like to learn all about setting up servers with Ubuntu Server. This book assumes that the reader knows the basics of Linux, such as editing configuration files and running basic commands.

## What this book covers

*Chapter 1*, *Deploying Ubuntu Server*, covers the installation process for Ubuntu Server. This chapter walks you through creating bootable media and the installation process.

*Chapter 2*, *Managing Users and Permissions*, covers user management in full. Topics here will include creating and removing users, configuring password policies, and using the `sudo` command, as well as group management and switching from one user to another.

*Chapter 3*, *Managing Software Packages*, takes the reader through the process of searching for, installing, and managing packages. This will include managing APT repositories and installing packages, and even a look at Snap packages.

*Chapter 4*, *Navigating and Essential Commands*, teaches you the essential commands needed necessary for navigating through directory trees, viewing the contents of log files, and perusing log files.

*Chapter 5*, *Managing Files and Directories*, expands on the knowledge gained from the previous chapter and rounds out your toolset of essential commands by going over how to edit, copy, move, and rename files.

*Chapter 6*, *Boosting Your Command-line Efficiency*, goes over additional tips, tricks, and techniques to enhance the reader's usage of the command line. Topics here include managing output, investigating Bash history, and more.

*Chapter 7*, *Controlling and Managing Processes*, teaches the reader how to manage what is running on the server, as well as how to stop misbehaving processes. This will include having a look at htop, systemd, and managing jobs.

*Chapter 8*, *Monitoring System Resources*, goes over how to manage valuable system resources on your server, such as viewing disk and memory usage, as well as understanding load average and how it impacts your CPU.

*Chapter 9*, *Managing Storage Volumes*, takes a look at storage volumes. You'll be shown how to view disk usage, format volumes, manage the `/etc/fstab` file, use LVM, and more. In addition, we'll look at managing swap.

*Chapter 10*, *Connecting to Networks*, takes a look at networking in Ubuntu, specifically how to connect to resources from other nodes. We'll look at assigning IP addresses, connecting to other nodes via OpenSSH, and name resolution.

*Chapter 11*, *Setting Up Network Services*, revisits networking with more advanced concepts. In this chapter, the reader will learn more about the technologies that glue our network together, such as DHCP and DNS. The reader will set up their own DHCP and DNS server, as well as installing NTP.

*Chapter 12*, *Sharing and Transferring Files*, is all about sharing files with others. Concepts will include the setup of Samba and NFS network shares, and we will even go over transferring files manually with rsync and scp.

*Chapter 13*, *Managing Databases*, takes the reader through the journey of setting up and managing databases via MariaDB. The reader will learn how to install MariaDB, how to set up databases, and how to create a secondary database server.

*Chapter 14*, *Serving Web Content*, takes a look at serving content with Apache. In addition, the reader will be shown how to secure Apache with an SSL certificate, manage modules, and set up keepalived. Installing Nextcloud is also covered.

*Chapter 15*, *Automating Server Configuration with Ansible*, will show the reader how to set up a Git repository for holding configuration management scripts, how to use the powerful Ansible tool to automate common administrative tasks, and also how to use ansible-pull.

*Chapter 16*, *Virtualization*, is all about virtualization (unsurprisingly!). The reader will be walked through setting up their very own KVM installation, as well as how to manage virtual machines with virt-manager.

*Chapter 17*, *Running Containers*, discusses the subject of containers and shows the reader how to manage containers in both Docker and LXD.

*Chapter 18*, *Container Orchestration*, teaches you how to take containers to the next level and manage them with the power of Kubernetes. You'll not only learn how to install Micro K8s, but also how to build your very own Kubernetes cluster from scratch.

*Chapter 19*, *Deploying Ubuntu in the Cloud*, shows you how to spin up Ubuntu servers in the cloud, with an introduction to **Amazon Web Services** (**AWS**).

*Chapter 20*, *Automating Cloud Deployments with Terraform*, goes over the process of automating the process of building cloud infrastructure using Terraform.

*Chapter 21*, *Securing your Server*, takes a look at various things the reader can do to strengthen security on Ubuntu servers. Topics will include lowering the attack surface, securing OpenSSH, setting up a firewall, and more.

*Chapter 22*, *Troubleshooting Ubuntu Servers*, consists of topics relating to things we can do when our deployments don't go according to plan. The reader will also investigate the problem space, view system logs, and trace network issues.

*Chapter 23*, *Preventing Disasters*, informs the reader of various strategies that can be used to prevent and recover from disasters. This includes a look at utilizing Git for configuration management, implementing a backup plan, and more.

## To get the most out of this book

This book is for readers who already have some experience with Linux, though it doesn't necessarily have to be with Ubuntu. Preferably, the reader will understand basic Linux command-line skills, such as changing directories, listing contents, and issuing commands as regular users or with root. Even if you don't have these skills, you should read this book anyway—the opening chapters will cover many of these concepts.

In this book, we'll take a look at real-world situations in which we can deploy Ubuntu Server. This will include the installation process, serving web pages, setting up databases, and much more. Specifically, the goal here is to be productive. Each chapter will teach the reader a new and valuable concept, using practical examples that are relative to real organizations. Basically, we focus on getting things done, rather than focusing primarily on theory. Although the theory that goes into Linux and its many distributions is certainly interesting, the goal here is to get you to the point where if a work colleague or client asks you to perform work on an Ubuntu-based server, you'll be in a good position to get the task done. Therefore, if your goal is to get up and running with Ubuntu Server and learn the concepts that really matter, this book is definitely for you.

To follow along, you'll either need a server on which to install Ubuntu Server, a virtual Ubuntu instance from a cloud provider, or a laptop or desktop capable of running at least one virtual machine.

## Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Ubuntu-Server_Third-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781800564640_ColorImages.pdf`.

## Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Anyway, back to our `/etc/passwd` file. The fifth column is designated for user info, most commonly the user's first and last names."

A block of code is set as follows:

```
description: External access profile
devices:
  eth0:
  name: eth0
  nictype: bridged
  parent: br0
  type: nic
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are highlighted:

```
description: External access profile
devices:
  eth0:
  name: eth0
  nictype: bridged
  parent: br0
  type: nic
```

Any command-line input or output is written as follows:

```
sudo apt install docker.io
```

**Bold**: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "LXC (pronounced Lex-C) is short for **Linux Containers** and is another implementation of containerization, very similar to Docker."

Warnings or important notes appear like this.

Tips and tricks appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback**: Email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

**Errata**: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy**: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

**If you are interested in becoming an author**: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

# 1
# Deploying Ubuntu Server

Ubuntu Server is an extremely powerful distribution of Linux for servers and network appliances. Whether you're setting up a high-end database or a small office file server, the flexible nature of Ubuntu Server will meet and surpass your needs. In this book, we'll walk through all the common use cases to help you get the most out of this exciting platform. Ubuntu Server features a perfect mix of modern development frameworks and rock-solid stability, and its hardware support enables it to be installed on the latest server hardware.

In this chapter, I'll guide you through the process of deploying Ubuntu Server from start to finish. We'll begin with some discussion of best practices, and then we'll obtain the software and create our installation media. Next, I'll give you a step-by-step rundown of the entire installation procedure. By the end of this chapter, you'll have an Ubuntu Server installation of your own to use throughout the remainder of this book. In addition, since Canonical (the makers of Ubuntu) now offer full support for Raspberry Pi with Ubuntu 20.04, we'll look at the process of setting that up as well.

In this chapter, we will cover:

- Technical requirements
- Determining your server's role
- Setting up our server
- Obtaining installation media
- Creating a bootable flash drive
- Installing Ubuntu Server
- Installing Ubuntu Server on a Raspberry Pi

To get started, we'll first take a look at some of the technical requirements for deploying a server with Ubuntu.

## Technical requirements

To follow along with the examples in this book, you'll need an Ubuntu Server installation to work with. In general, the following specifications are the estimated minimums to successfully install Ubuntu Server:

- 64-bit CPU
- 1GB RAM
- 10GB hard disk (16GB or more is recommended)

64-bit CPU support is now a requirement, as Canonical no longer makes versions of Ubuntu available for 32-bit processors (with the only exception being older models of the Raspberry Pi). While this may seem like a surprising requirement, all computers sold today support 64-bit operating systems, and consumer CPUs have been 64-bit capable since at least 2003. Even if you have an older PC lying around that you don't think is capable of running a 64-bit operating system, you'd be surprised—even the later models of the Pentium IV (which is quite old) supports this, so this requirement shouldn't be hard to meet. Don't worry about the particulars of this right now, we'll go through the requirements in more detail later on in this chapter.

Now that we understand the technical requirements of Ubuntu Server, let's consider the role our server will play in our organization.

## Determining your server's role

While at this point your goal is most likely to set up an Ubuntu Server installation for the purposes of following along with the examples contained within this book, it's also important to understand how a typical server rollout is performed in the real world. Every server must have a purpose, or *role*. This role could be that of a database server, web server, file server, and so on. In a nutshell, the role is the value the server adds to you or your organization. Sometimes, servers may be implemented solely for the purpose of testing experimental code. And this is important too—having a test environment is a very common (and worthwhile) practice.

Once you understand the role your server plays within your organization, you can plan for its implementation. Is the system mission critical? How would it affect your organization if for some reason this server malfunctioned? Depending on the answer to this question, you may only need to set up a single server for this task, or you may wish to plan for redundancy such that the server doesn't become a central point of failure. An example of this may be a **DNS server**, which would affect your colleagues' ability to resolve local hostnames and access required resources. It may make sense to add a second DNS server to take over in the event that the primary server becomes unavailable for some reason.

Another item to consider is how confidential the data residing on a server is going to be for your environment. This directly relates to the installation procedure we're about to perform, because you will be asked whether or not you'd like to utilize **encryption**. The encryption that Ubuntu Server offers during installation is known as **encryption at rest**, which refers to the data stored within the internal storage volumes on that server. If your server is destined to store confidential data (accounting information, credit card numbers, employee or client records, and so on), you may want to consider making use of this option. Encrypting your hard disks is a really good idea to prevent miscreants with local access from stealing data. As long as the attacker doesn't have your encryption key, they cannot steal this confidential information. However, it's worth mentioning that anyone with physical access can easily destroy data (encrypted or not), so remember to keep your server room locked!

At this point in the book, I'm definitely not asking you to create a detailed implementation diagram or anything like that, but instead to keep in mind some concepts that should always be part of the conversation when setting up a new server. It needs to have a reason to exist, it should be understood how critical and confidential the server's data will be, and the server should then be set up accordingly. Once you practice these concepts as well as the installation procedure, you can make up your own server roll-out plan to use within your organization going forward. All in all, understanding the purpose of each component in your infrastructure is a great mindset to adopt.

At this point, we now understand how we might identify a role for our server and how it may fit in with our organization. In the next section, we'll take a look at the process of actually installing Ubuntu Server, so we will have at least one test machine to use for the examples in this book.

## Setting up our server

Now it's time to set up an installation of Ubuntu Server to use for the examples in this book. But before we can do that, we have to decide what to actually install it on. For the purposes of this book, there isn't a specific requirement in terms of hardware. You just need an Ubuntu Server installation of some sort, and it wouldn't hurt to set up multiple servers if you can—you don't need them all to be on the same device type. Having multiple servers will help you experiment with networking when we get to that point later on in the book. But for now, it's only a matter of utilizing whatever you have at your disposal to get an Ubuntu installation going.

In particular, the following list includes the most common devices you can consider for your Ubuntu Server installation:

- Virtual machine
- Physical server
- Virtual private server
- Spare desktop or laptop
- Raspberry Pi

Let's take a look at each of these options in more detail.

## Virtual machine

Most computers sold nowadays support **Virtual Machines** (**VM**s). **VirtualBox** is a great solution, as it's easy to use and available for all of the major operating systems. Just like Ubuntu itself, VirtualBox is available for free, so it's typically the lowest-cost entry-point for getting started. Also, VirtualBox allows you to easily create snapshots of your Ubuntu installation, so you can create a point-in-time backup before going through an example in this book, and then restore it to repeat tasks as often as you'd like. The downside to VirtualBox is that you'll need to be able to dedicate at least 1 GB of RAM to your Ubuntu Server VM, and your CPU will need to support virtualization extensions, which you'll need to enable in your computer's settings if your device supports it.

VirtualBox can be downloaded here: `https://www.virtualbox.org`.

## Physical server

Nowadays, it's very easy to find used physical servers for an affordable price. Dell PowerEdge is a very common model, and the R610 and R710 specifically are good choices that are readily available. These servers are commonly made available in the reseller market after companies upgrade to newer models. The R610 and R710 are a bit old, but their specs are still great for testing purposes. The downside with physical servers is that they take up a lot of room and can often be power-hungry. Make sure to shut them down when not in use and look into the cost of electric services in your area—these servers can be very cheap to run or very expensive, depending on your electricity rates.

## Virtual private server

Services such as Amazon Web Services, Google Cloud, Linode, Microsoft Azure, Digital Ocean, and others allow you to set up Ubuntu Server in the cloud for you to connect to and manage via **OpenSSH**. Choosing the **Virtual Private Server** (**VPS**) option has some benefits; you don't need to find room for a large physical server, and you don't need to worry about power usage either. Another benefit is that you won't even need to go through the installation process in this chapter at all; the cloud provider will do that for you when you choose to deploy Ubuntu Server on their platform. The main downside, though, is that VPSes are not free—you'll need to look at the costs associated with running Ubuntu on such a server and decide if that cost makes sense. Some VPS services allow you to set up instances for as little as 5 USD per month, which can be lower than the cost of the electricity needed to run a physical server in some areas.

## Spare desktop or laptop

Did you just buy a new computer? Don't throw the old one away, you can re-purpose it! Spare desktops and laptops make great devices for testing Ubuntu Server. If the device is a laptop, then you benefit from a built-in **Uninterruptible Power Supply** (**UPS**) if the battery still holds a charge. With a built-in screen and keyboard, as well as lower power usage, these can be a convenient choice for acting as a test server. In fact, laptops and desktops typically support virtualization, so you'll be able to set up a VM server on these, and running containers (such as with **Docker**) isn't outside the realms of possibility either.

## Raspberry Pi

**Raspberry Pi** units are quickly becoming a user favorite for several use cases. They are inexpensive (some models are available for less than 50 USD) and they also use very minimal electricity—you can leave them powered on 24/7 with a virtually unnoticeable difference in your power bill. In fact, they use about as much power as it takes to charge a high-end cellphone. Another benefit is that a Raspberry Pi is generally going to be more powerful than an entry-level VPS. The cheapest VPS instances typically have just 1 CPU core and 1 GB of RAM, but modern Raspberry Pis are produced with quad-core CPUs, as well as 2 GB, 4 GB, or 8 GB of RAM depending on which model you purchase. This means that a Raspberry Pi can possibly perform better than cheaper VPS instances. The downside to Raspberry Pi is that some applications are unavailable, since they utilize ARM CPUs instead of x86. This means that some examples in this book will not work on a Pi (although the majority will).

Once you've chosen a device to install Ubuntu Server on, we can continue. If you chose to utilize a VPS, you can move on to the next chapter, *Chapter 2*, *Managing Users and Permissions*, as you won't need to walk through the installation process. In the case of the Raspberry Pi, if that's your chosen platform, you can skip to the end of this chapter for a dedicated section about setting that up. For all other devices, continue reading for a walk-through for the Ubuntu Server Live Installer.

## Obtaining installation media

It's time to get started! If you've decided to utilize a physical server, desktop, laptop, or VM as your test server, then you'll need to go through the installation process to set up Ubuntu. Don't worry—it's very easy to do and is made even easier in Ubuntu 20.04 as there are fewer overall steps in the process. If you've instead opted to use a VPS or Raspberry Pi, you won't need to go through this process, as VPS providers do this for you and Raspberry Pi has a different setup method altogether (we'll cover this later in the chapter, in the *Installing Ubuntu on a Raspberry Pi* section).

Assuming that you've decided to use a device that does require going through the installer, we'll need to download Ubuntu Server and then create bootable installation media to install it. How you do this largely depends on your hardware. Does your device have an optical drive? Is it able to boot from USB? Refer to the documentation for your device to find out.

It's recommended to utilize a flash drive for the installation if you can, preferably one that uses USB 3.0 or higher since you'd benefit from its faster speed compared to USB 2.0. The reason for the preference toward using a flash drive is due to the fact that they are typically faster than a DVD.

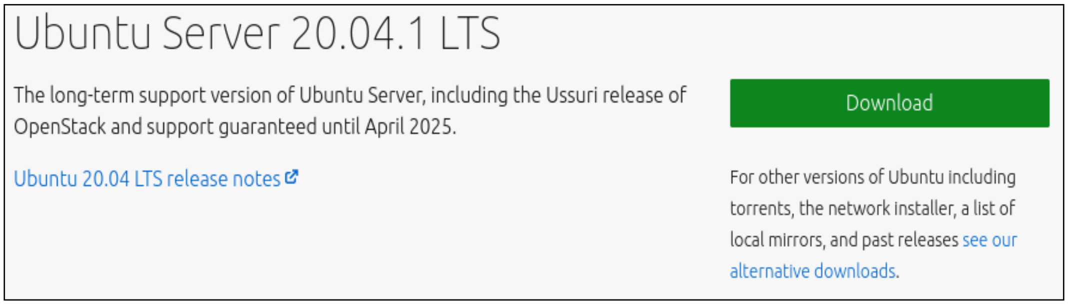
However, if your device is older, you won't have a choice, as legacy devices were not able to boot from USB at all. As a general rule, use a flash drive if you can and opt for a DVD only if you have no choice.

In the past, Ubuntu Server ISO images could be used to create either a bootable CD or DVD. Nowadays, writable CDs don't have enough space to support the download size. Therefore, if you choose to burn bootable optical media, you'll need a writable DVD at a minimum.

Unfortunately, the differing age of servers within a typical data center introduces some unpredictability when it comes to how to boot installation media. When I first started with servers, it was commonplace for all standard rack servers to contain a 3.5-inch floppy disk drive, and some of the better ones even contained an optical drive. Nowadays, servers typically contain neither. If a server does have an optical drive, it could potentially go unused for an extended period of time and become faulty without anyone knowing until the next time someone goes to use it. Some servers boot from USB, others don't. To continue, check the documentation for your hardware and plan accordingly. Your server's capabilities will determine which kind of media you'll need to create.

Regardless of whether we plan on creating a bootable USB or DVD, we only need to download a single file. Navigate to the following site in your web browser to get started: `https://www.ubuntu.com/download/server`.

From this page, we're going to download Ubuntu 20.04 LTS by clicking on the **Download** button:



Ubuntu Server 20.04.1 LTS

The long-term support version of Ubuntu Server, including the Ussuri release of OpenStack and support guaranteed until April 2025.

Ubuntu 20.04 LTS release notes

For other versions of Ubuntu including torrents, the network installer, a list of local mirrors, and past releases see our alternative downloads.

Figure 1.1: Ubuntu Server 20.04.1 download page

There may be other versions of Ubuntu Server listed on this page, such as 20.10 and 21.04, depending on when you're reading this. New releases of Ubuntu are published every six months. However, this book only covers the **Long Term Support** (**LTS**) release, due to the fact that it benefits from five years of support (non-LTS versions are only supported for nine months).

Organizations don't typically utilize non-LTS releases at all, except for testing upcoming features prior to general availability, so for our purposes, we'll stick with the LTS version. Once the download is completed, we'll end up with an ISO image we can use to create our bootable installation media.

If you're setting up a VM, then the ISO file you download from the Ubuntu downloads page will be all you need; you won't need to create a bootable DVD or flash drive. In that scenario, all you should need to do is create a VM, attach the ISO to the virtual optical drive, and boot it. From there, the installer should start, and you can proceed with the installation procedure outlined later in this chapter, in the *Installing Ubuntu Server* section. Going over the process of booting an ISO image on a VM differs from one virtualization solution to another, so detailing the process on each would be beyond the scope of this book. Thankfully, the process is usually straightforward and you can find the details within the documentation of your hypervisor or from performing a quick Google search. In most cases, the process is as simple as attaching the downloaded ISO image to the VM and then starting it up.

If your device does not support booting from USB and you find yourself needing to create a bootable DVD, the process is typically just a matter of downloading the ISO file and then right-clicking on it. In the right-click menu of your operating system, you should have an option to burn to disk or some similar verbiage. This is true of Windows, as well as most graphical desktop environments of Linux where a disk-burning application is installed.

The exact procedure differs from system to system, mainly because there is a vast amount of software combinations at play here. For example, I've seen many Windows systems where the right-click option to burn a DVD was removed by an installed CD/DVD-burning application. In that case, you'd have to first open your CD/DVD-burning application and find the option to create media from a downloaded ISO file. As much as I would love to outline the complete process here, no two Windows PCs typically ship with the same CD/DVD-burning application. The best rule of thumb is to try right-clicking on the file to see whether the option is there, and, if not, refer to the documentation for your application. Keep in mind that a *data disk* is not what you want, so make sure to look for the option to create media from an ISO image or your disk will be useless.

At this point, you should have an Ubuntu Server ISO image file downloaded. If you are planning on using a DVD to install Ubuntu, you should have that created as well. In the next section, I'll outline the process of creating a bootable flash drive that can be used to install Ubuntu Server.

## Creating a bootable flash drive

The process of creating a bootable USB flash drive with which to install Ubuntu used to vary greatly between platforms. The steps were very different depending on whether your workstation or laptop was currently running Linux, Windows, or macOS. Thankfully, a much simpler method has come about. Nowadays, I recommend the use of **Etcher** to create your bootable media. Etcher is fantastic in that it abstracts the method such that it is the same regardless of which operating system you use, and it distills the process to its most simple form.

Another feature I like is that Etcher is safe; it prevents you from destroying your current operating system in the process of mastering your bootable media. In the past, you'd use tools like the `dd` command on Linux to write an ISO file to a flash drive. However, if you set up the `dd` command incorrectly, you could effectively write the ISO file over your current operating system and wipe your entire hard drive. Etcher doesn't let you do that.

Before continuing, you'll need a USB flash drive that is either empty or one you don't mind wiping. This process will completely erase its contents, so make sure the device doesn't have information on it that you need. The flash drive should be at least 1 GB, preferably 2 GB or larger. Considering it's difficult to find a flash drive for sale with less than 4 GB of space nowadays, this should be relatively easy to obtain.

To get started, head on over to `https://www.balena.io/etcher/`, download the latest version of the application from their site, and open it up. The window will look similar to the following screenshot once it launches:

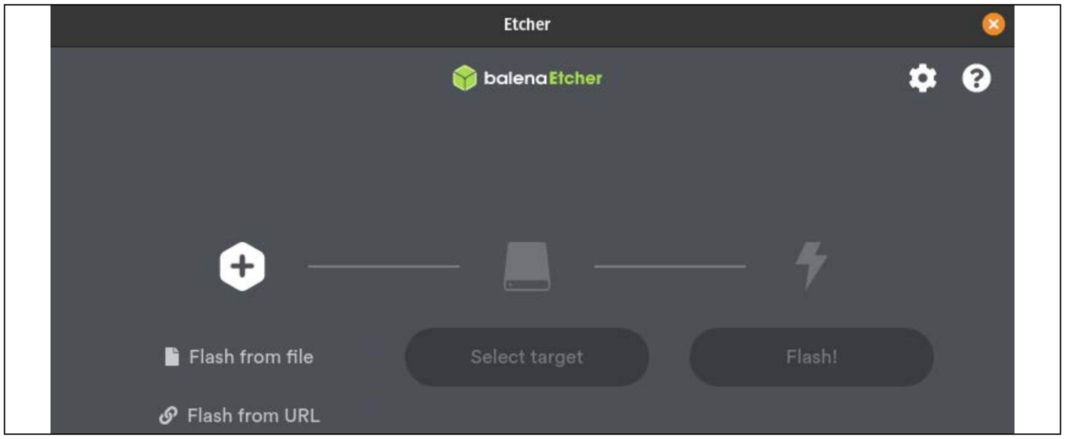
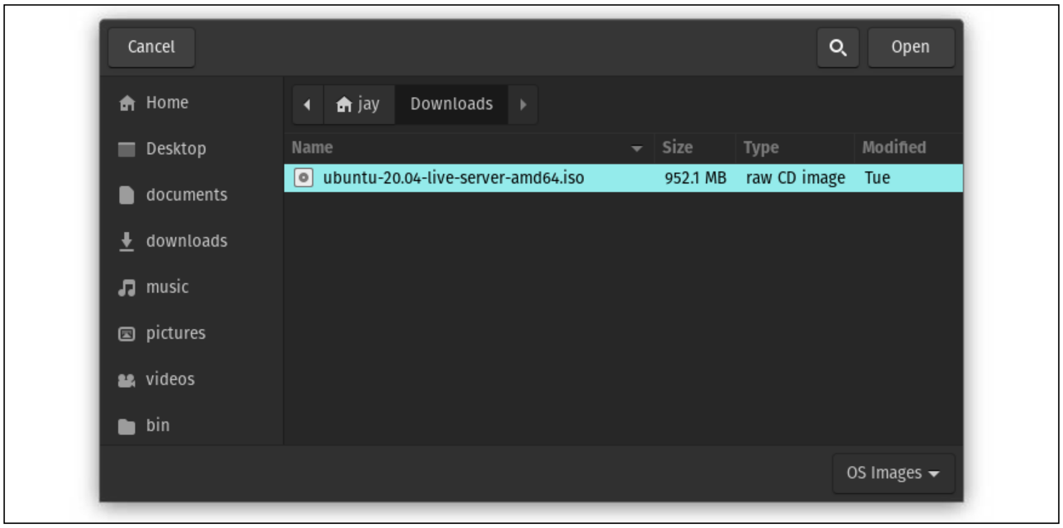


Figure 1.2: Utilizing Etcher to create a bootable flash drive

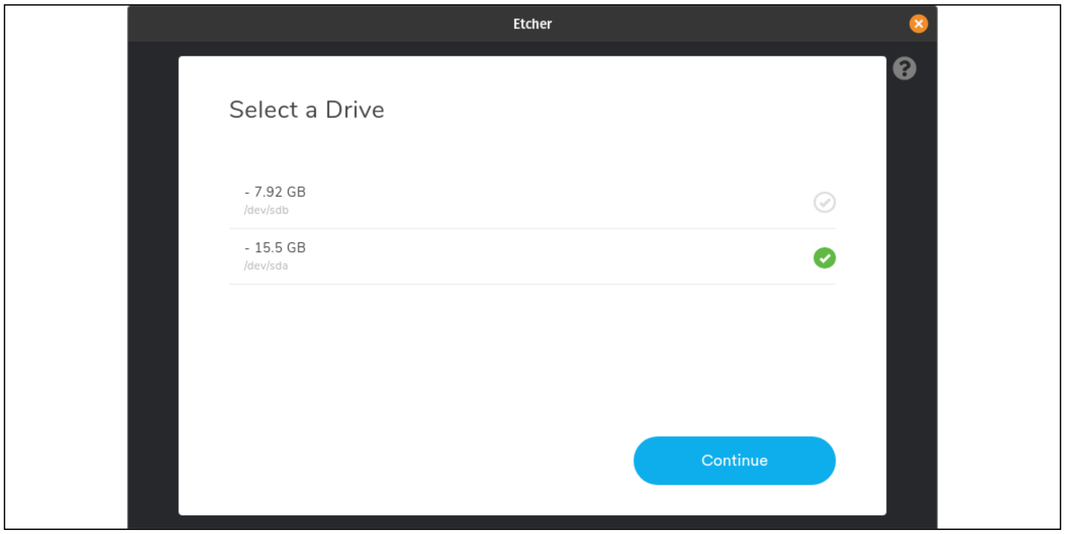
At this point, click **Flash from file**, which will open up a new window that will allow you to select the ISO file you downloaded earlier. Once you select the ISO, click on **Open**:



ubuntu-20.04-live-server-amd64.iso 952.1 MB raw CD image Tue

Figure 1.3: Selecting an ISO image with Etcher

If your flash drive is already inserted into the computer, Etcher should automatically detect it. In the event that you have more than one flash drive attached, or Etcher selects the wrong one, you can click **Change** and select the flash drive you wish to use:



Select a Drive

- 7.92 GB
/dev/sdb

- 15.5 GB ✓
/dev/sda

Figure 1.4: Selecting a different flash drive with Etcher

Finally, click **Continue** to get the process started. At this point, the flash drive will be converted into Ubuntu Server installation media that can then be used to start the installation process:

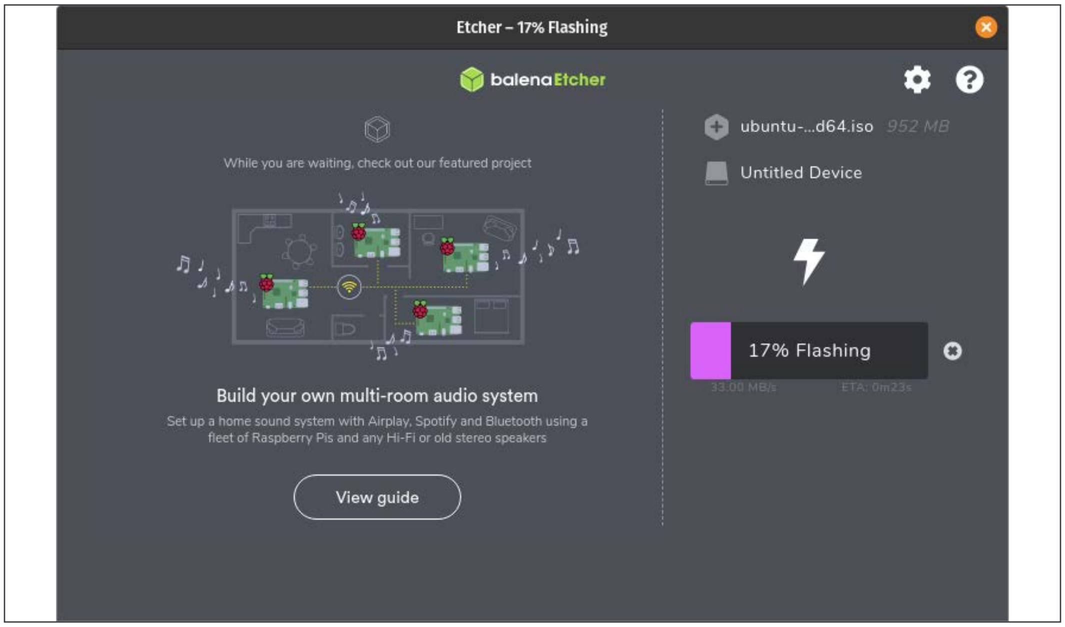

Etcher – 17% Flashing
balenaEtcher
While you are waiting, check out our featured project
Build your own multi-room audio system
Set up a home sound system with Airplay, Spotify and Bluetooth using a fleet of Raspberry Pis and any Hi-Fi or old stereo speakers
View guide
ubuntu-...d64.iso 952 MB
Untitled Device
17% Flashing
33.00 MB/s
ETA: 0m23s

Figure 1.5: Etcher in the progress of writing a flash drive

After a few minutes (the length of time varies depending on your hardware), the **Flashing** process will complete, and you'll be able to continue and get some installations going. Before we get into that, though, we should have a quick discussion regarding partitioning.

## Planning the partitioning layout

**Partitioning** your disk allows you to divide up your hard disk to dedicate specific storage allocations to individual applications or purposes. For example, you can dedicate a partition for the files that are shared by your Apache web server so changes to other partitions won't affect it. You can even dedicate a partition for your network file shares—the possibilities are endless. Each partition is mounted (attached) to a specific directory, and any files sent to that directory are thereby sent to that separate partition. The names you create for the directories where your partitions are mounted are arbitrary; it really doesn't matter what you call them. The flexible nature of storage on Linux systems allows you to be creative with your partitioning as well as your directory naming, since the Linux filesystem gives you more flexibility when it comes to mounting storage devices in specific folders.

Admittedly, we're getting ahead of ourselves here. After all, we're only just getting started and the point of this chapter is to help you set up a basic Ubuntu Server installation to serve as the foundation for the rest of the chapter. When going through the installation process, we'll accept the defaults anyway. However, the goal of this section is to give you examples of the options you have for consideration later. At some point, you may want to get creative and play around with the partition layout.

With custom partitioning, you're able to do some very clever things. For example, with the right configuration, you'll be able to wipe and reload your distribution while preserving user data, logs, and more. This works because Ubuntu Server allows you to carve up your storage any way you want during installation. If you already have a partition with data on it, you can choose to leave it as is so you can carry it forward into a new install. You simply set the directory path where it's mounted to be the same as before, restore your configuration files, and your applications will continue working as if nothing happened.

One very common example of custom partitioning in the real world is separating the `/home` directory into its own partition. Since this is where users typically store their files, you can set up your server such that a reload of the distribution won't disturb their files. When they log in after a server refresh, all their files will be right where they left them. You could even place the files shared by your Apache web server on to their own partition and preserve those too. You can get very creative here.

> It probably goes without saying, but when reinstalling Ubuntu, you should back up partitions that have data you don't want to be wiped (even if you don't plan on formatting the partitions). The reason being, one wrong move (literally a single checkbox) and you can easily wipe out all the data on that partition. Always back up your data when refreshing a server.

Another reason to utilize separate partitions may be to simply create boundaries or restrictions. If you have an application running on your server that is prone to filling up large amounts of storage, you can point that application to its own partition, limited by size. An example of where this could be useful is an application's log files. Log files are the bane of any system administrator's life when it comes to storage. While helpful if you're trying to figure out why something crashed, logs can fill up a hard disk very quickly if you're not careful. In my experience, servers have been known to come to a screeching halt due to log files filling up all the available free space on a server where everything was on a single partition. The only boundary the application had was the entirety of the disk itself.

While there are certainly better ways of handling excessive logging (log rotating, disk quotas, and so on), a separate partition would certainly help. If the application's log directory was on its own partition, it would be able to fill up that partition, but not the entire drive, which would cause issues but wouldn't affect the entire server. As an administrator, it's up to you to weigh the pros and the cons of these strategies to avoid server overload and develop a partitioning scheme that will best serve the needs of your organization.

Success when maintaining a server is a matter of efficiently managing resources, users, and security—a good partitioning scheme is certainly part of that. Sometimes it's just a matter of making things easier on yourself so that you have less work to do should you need to reload your operating system. For the sake of following along with this book, it really doesn't matter how you install or partition Ubuntu Server. The trick is simply to get it installed—you can always practice partitioning later. After all, part of learning is setting up a platform, observing any issues that may occur, and then fixing it up.

Here are some basic tips regarding partitioning:

- At a minimum, a partition for the root filesystem (designated by a forward slash) is required.
- The `/var` directory contains most log files and is therefore a good candidate for separation for the reasons mentioned previously in this section.
- The `/home` directory stores all user files. Separating this into a separate partition can be beneficial as it gives you the possibility of having your user files survive a reinstall of Ubuntu.
- If you've used Linux before, you may be familiar with the concept of a **swap partition**, which is a special partition that can act as RAM when your memory becomes full. This is no longer necessary—a **swap file** will be created automatically in newer Ubuntu releases.

When we perform our installation in the next section, we'll choose the defaults for the partitioning scheme to get you started quickly. However, I recommend you come back to the installation process at some point in the future and experiment with it. You may come up with some clever ways to split up your storage. However, you don't have to—having everything in one partition is fine too, depending on your needs.

Now that we have an understanding of what partitioning is, we should have all of the essential topics covered to enable us to actually get the installation of Ubuntu Server going. Let's go ahead and work on that now.

## Installing Ubuntu Server

At this point, we should be ready to get an installation of Ubuntu Server going. In the steps that follow, I'll walk you through the process.

### Installing media

To get started, all you should need to do is insert the installation media into your server or device and follow the onscreen instructions to open the boot menu. The key you press at the beginning of the POST process differs from one machine to another, but it's quite often *F10*, *F11*, or *F12*. Refer to your documentation if you are unsure, though most computers and servers tell you which key to press at the beginning. You may miss this window of opportunity the first few times, and that's fine—to this day I still seem to need to restart the machine once or twice to hit the key in time.

When you first boot from the Ubuntu Server install media, you'll see an icon near the bottom that looks like the following (it will go away after a few seconds):

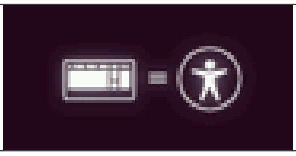


Figure 1.6: This icon indicates other options are available

This icon is your indication that you can press *Enter* here to select additional options, such as your language. If you don't press *Enter* within a few seconds, the installer will proceed automatically with the default options (such as setting the language to English). If you do press *Enter*, you'll see a screen something like the following screenshot:

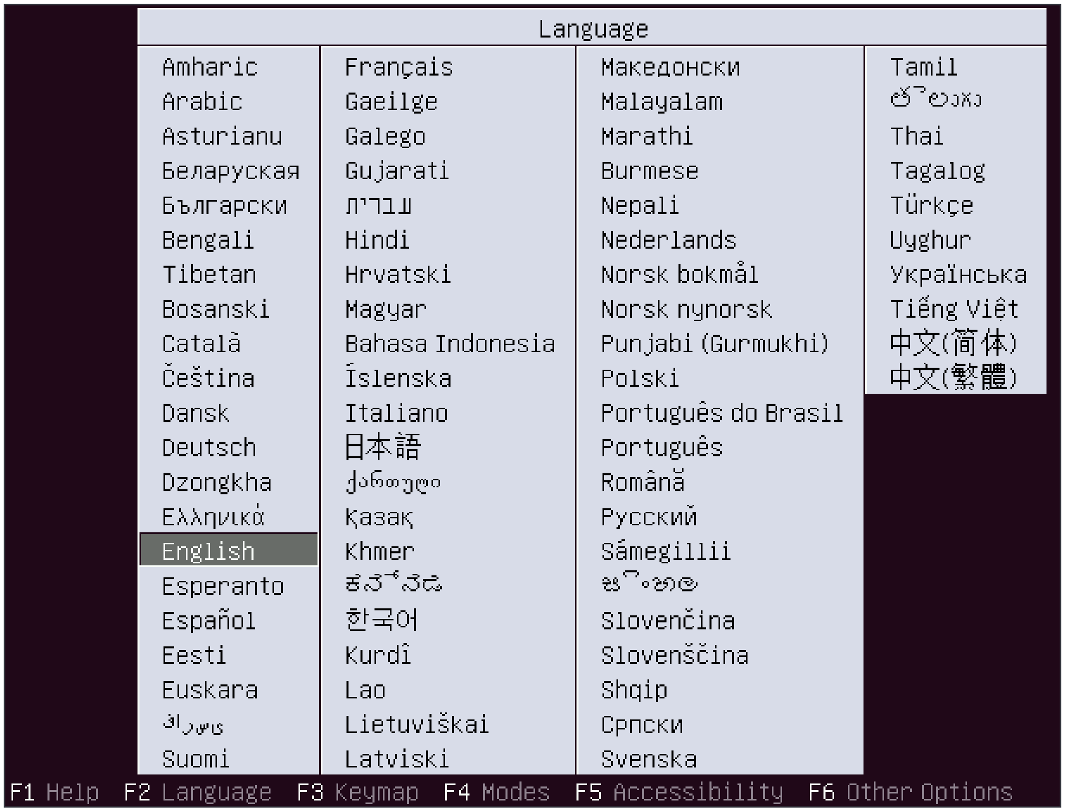


Figure 1.7: Language selection at the beginning of the installation process

If your language is anything other than English, you'll be able to select that here. You can also press *F6* to view **Other Options**, which you would only explore in a situation in which your hardware wasn't working properly. Most people won't need to do this, however.

After choosing your language, you'll be brought to the installation menu. (If you didn't press *Enter* when the previously mentioned icon was on the screen, this screen will be bypassed and you won't see it). The installation menu will give you additional options. To start the installation process, press *Enter* to choose the first option (**Install Ubuntu Server**), though a few of the other options may be useful to you:

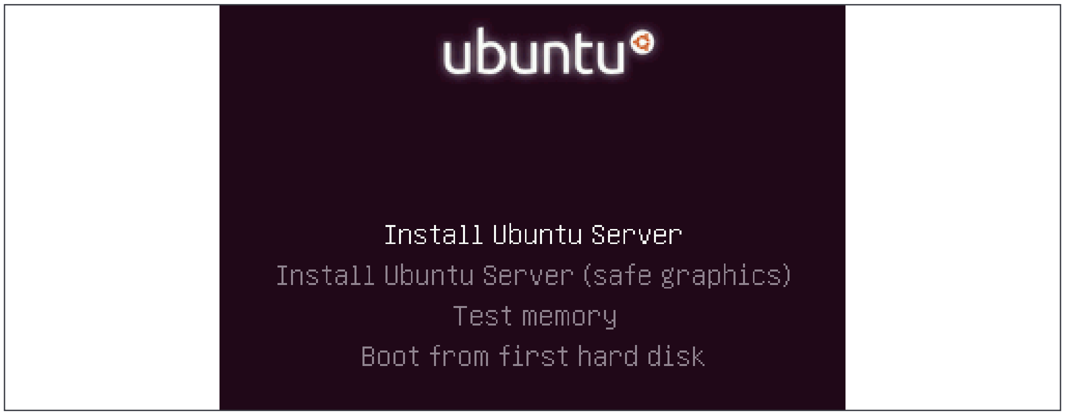
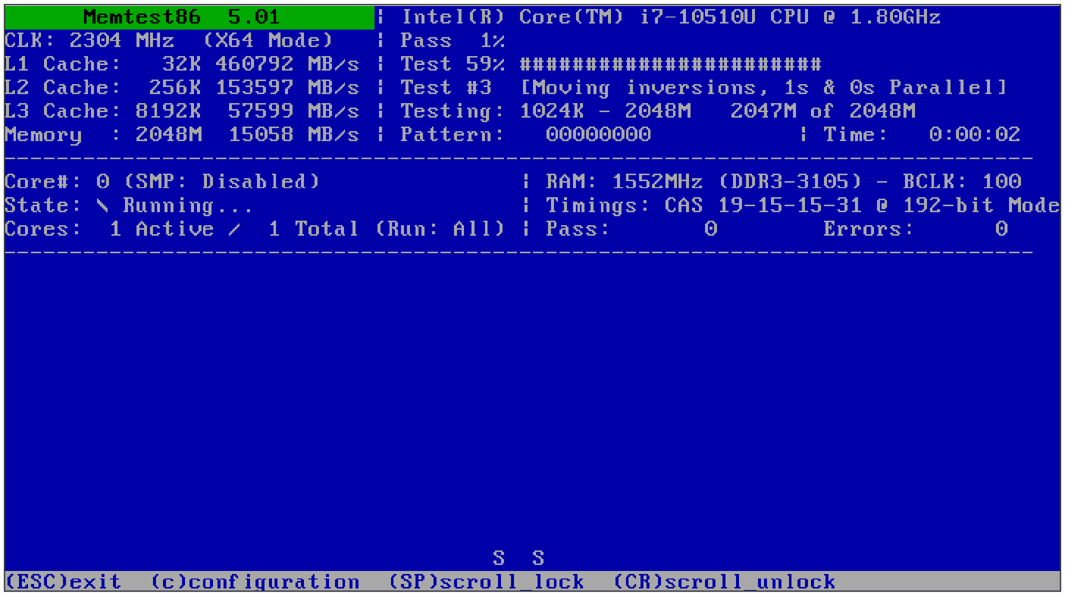

ubuntu
Install Ubuntu Server
Install Ubuntu Server (safe graphics)
Test memory
Boot from first hard disk

Figure 1.8: Main menu of the Ubuntu installer

First, this menu also offers you an option to install Ubuntu with **safe graphics** mode enabled. Normally, you shouldn't need this option. If you have any trouble with the installer (such as not being able to see anything on the screen after the installer starts), then this option may be something you'd want to try. If you aren't having any problems at all, you can ignore this.

Second, this menu gives you the option to **Test memory**. This is an option I've found myself using far more often than I'd like to admit. Defective RAM on computers and servers is surprisingly common and can cause all kinds of strange things to happen. For one, defective RAM may cause your installation to fail. Worse, your installation could also succeed—and I say that's worse due to the fact you'd find out about problems later (after spending considerable time setting it up) rather than right away. The following image shows a memory test in progress:



```
      Memtest86  5.01      | Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
CLK: 2304 MHz  (X64 Mode)  | Pass  1%
L1 Cache:   32K 460792 MB/s | Test 59% #######################
L2 Cache:  256K 153597 MB/s | Test #3  [Moving inversions, 1s & 0s Parallel]
L3 Cache: 8192K  57599 MB/s | Testing: 1024K - 2048M   2047M of 2048M
Memory  : 2048M  15058 MB/s | Pattern:   00000000          | Time:   0:00:02
------------------------------------------------------------------------------
Core#: 0 (SMP: Disabled)              | RAM: 1552MHz (DDR3-3105) - BCLK: 100
State: \ Running...                   | Timings: CAS 19-15-15-31 @ 192-bit Mode
Cores:  1 Active /  1 Total (Run: All) | Pass:        0        Errors:      0
------------------------------------------------------------------------------

                                     S  S
(ESC)exit  (c)configuration  (SP)scroll_lock  (CR)scroll_unlock
```

Figure 1.9: Memory test in progress

While doing a memory test can add considerable time to the installation process, I definitely recommend it on physical servers, especially if the hardware is new and hasn't been tested. In fact, I make it a practice to test the RAM of all my servers once or twice a year, so any installation media you create can be used as a memory tester any time you need one.

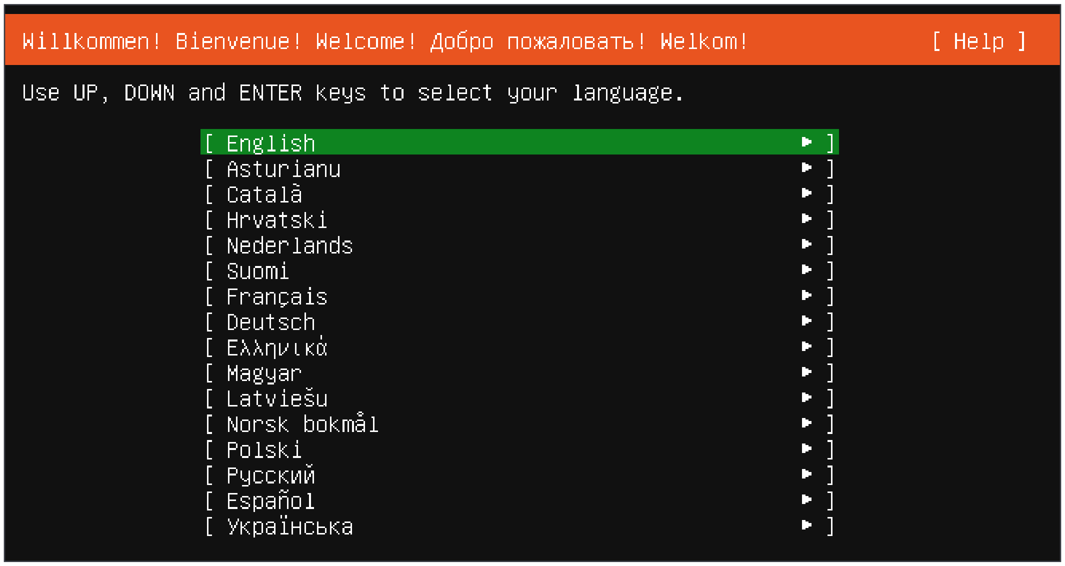
The memory test can take a very long time to complete (multiple hours) depending on how much RAM your server has. As a general rule of thumb, I've found that if there are issues with your server's memory, it will typically find the problem in fewer than 15 minutes. Press *Esc* to abort the test.

**Boot from first hard disk**, pictured in *Figure 1.8*, allows you to boot from the first hard disk, which is typically an existing operating system. If you left the installation media in the PC after the installation was finished, you can choose this option to attempt to boot the existing OS.

From this point forward, we will progress through the various screens to customize our installation. I'll guide you along the way.

## Customizing the installation

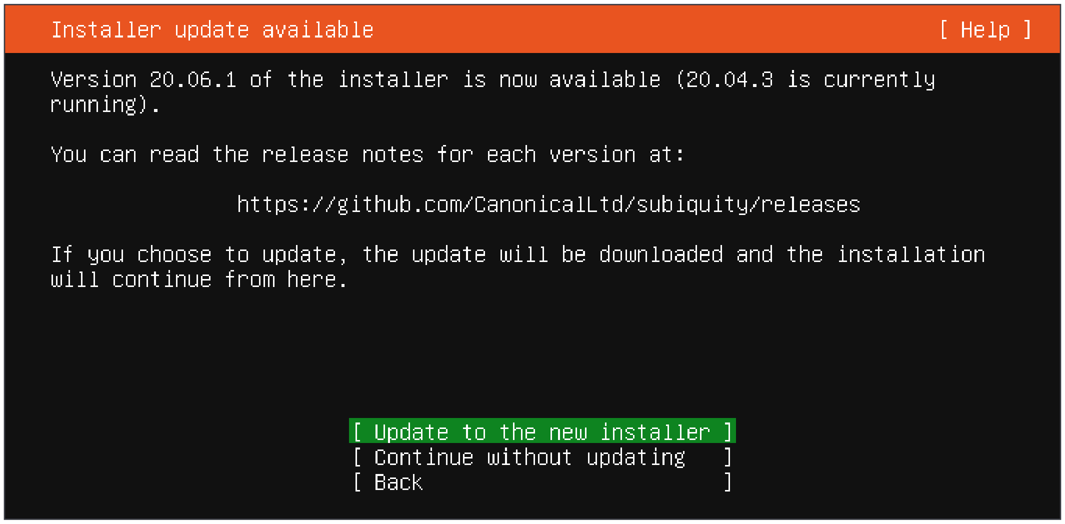
To navigate the installer, you simply use the arrow keys to move up and down to select different options, and press *Enter* to confirm choices. The *Esc* key will allow you to exit from a sub-menu. The installer is pretty easy to navigate once you get the hang of it. Anyway, after you enter the actual installation process, the first screen will ask you to select a language. Go ahead and do so:



```
Willkommen! Bienvenue! Welcome! Добро пожаловать! Welkom!          [ Help ]

Use UP, DOWN and ENTER keys to select your language.

              [ English                                  ▶ ]
              [ Asturianu                                ▶ ]
              [ Català                                   ▶ ]
              [ Hrvatski                                 ▶ ]
              [ Nederlands                               ▶ ]
              [ Suomi                                    ▶ ]
              [ Français                                 ▶ ]
              [ Deutsch                                  ▶ ]
              [ Ελληνικά                                 ▶ ]
              [ Magyar                                   ▶ ]
              [ Latviešu                                 ▶ ]
              [ Norsk bokmål                             ▶ ]
              [ Polski                                   ▶ ]
              [ Русский                                  ▶ ]
              [ Español                                  ▶ ]
              [ Українська                               ▶ ]
```

Figure 1.10: Language selection screen

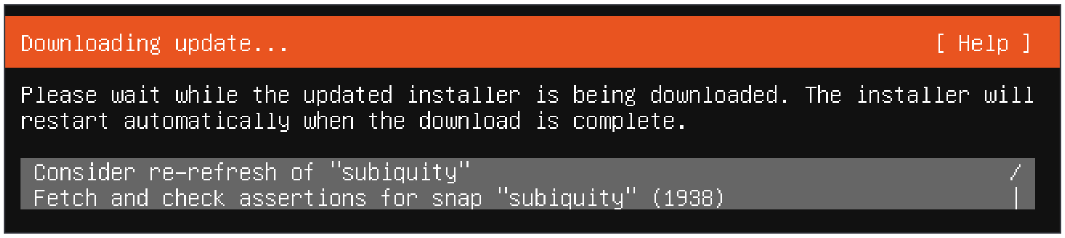
You may have noticed that the file name for the ISO file you downloaded included the term "live." At the next screen, you'll see why—if there's a new version of the installer available, you will see an offer to download the latest version. This will ensure that any bugs or vulnerabilities discovered later are fixed and won't become part of your installation. If you do see this option, it's highly recommended to allow it to download the fresh install program:



```
Installer update available                                        [ Help ]

Version 20.06.1 of the installer is now available (20.04.3 is currently
running).

You can read the release notes for each version at:

           https://github.com/CanonicalLtd/subiquity/releases

If you choose to update, the update will be downloaded and the installation
will continue from here.



                  [ Update to the new installer ]
                  [ Continue without updating   ]
                  [ Back                        ]
```

Figure 1.11: Installer update prompt

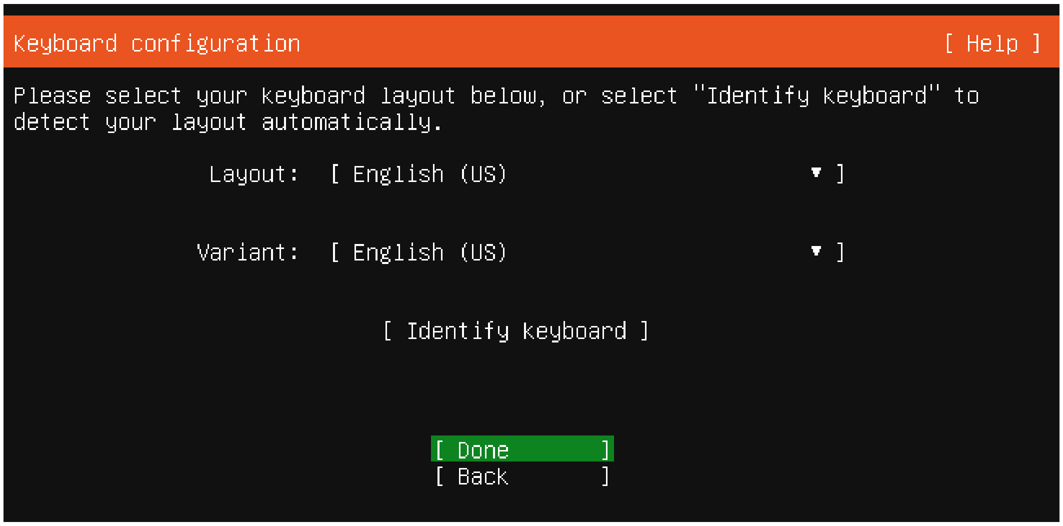
If an update is found and you agree to install it, the new installer should download very quickly and then you'll be good to go:



```
Downloading update...                                               [ Help ]

Please wait while the updated installer is being downloaded. The installer will
restart automatically when the download is complete.

 Consider re-refresh of "subiquity"                                         /
 Fetch and check assertions for snap "subiquity" (1938)                     |
```

Figure 1.12: Installing an installer update

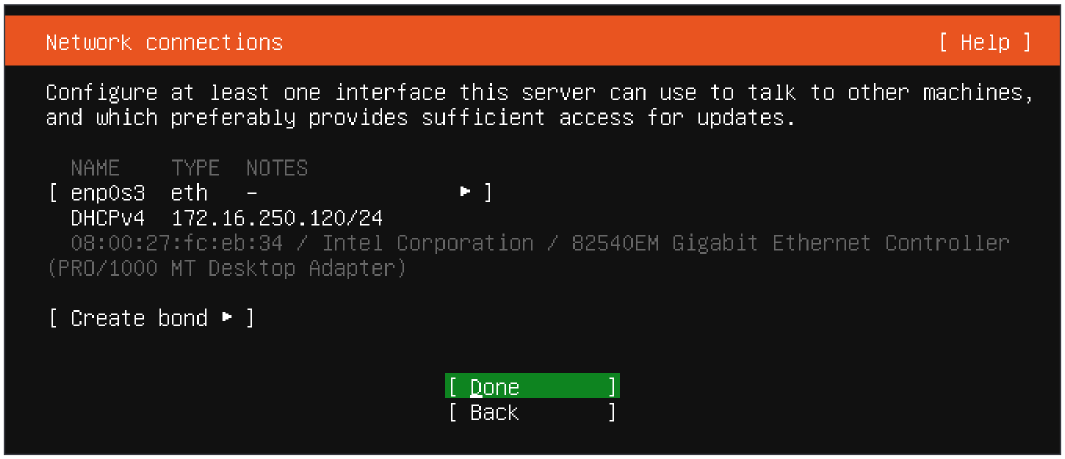
The next screen will allow you to choose your keyboard layout. If you use a keyboard other than an **English (US)** keyboard, you can select that here. Press the down key to highlight **Done** and press *Enter* when you've finished:



```
Keyboard configuration                                                [ Help ]
Please select your keyboard layout below, or select "Identify keyboard" to
detect your layout automatically.

               Layout:  [ English (US)                      ▾ ]


              Variant:  [ English (US)                      ▾ ]


                            [ Identify keyboard ]



                                 [ Done       ]
                                 [ Back       ]
```

Figure 1.13: Choosing a keyboard layout

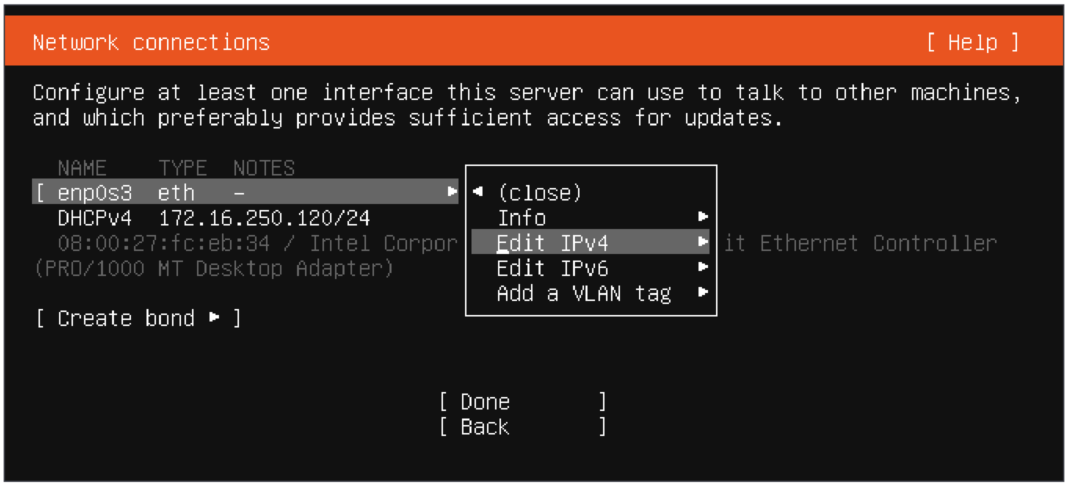
At the next screen, you'll be given a chance to set up your network connection. The default is set to DHCP, and you'll see an IP address here if the initial connection was successful. If your network doesn't automatically connect, follow the next few steps and change the connection to **Automatic (DHCP)**, as shown in *Figure 1.16*. If DHCP is acceptable to you, you can simply accept the defaults and select **Done** to continue along:



```
  Network connections                                                [ Help ]
  Configure at least one interface this server can use to talk to other machines,
  and which preferably provides sufficient access for updates.

    NAME    TYPE  NOTES
  [ enp0s3  eth   -                ▸ ]
    DHCPv4  172.16.250.120/24
    08:00:27:fc:eb:34 / Intel Corporation / 82540EM Gigabit Ethernet Controller
  (PRO/1000 MT Desktop Adapter)

  [ Create bond ▸ ]


                                 [ Done       ]
                                 [ Back       ]
```

Figure 1.14: Ubuntu Server network connection setup screen

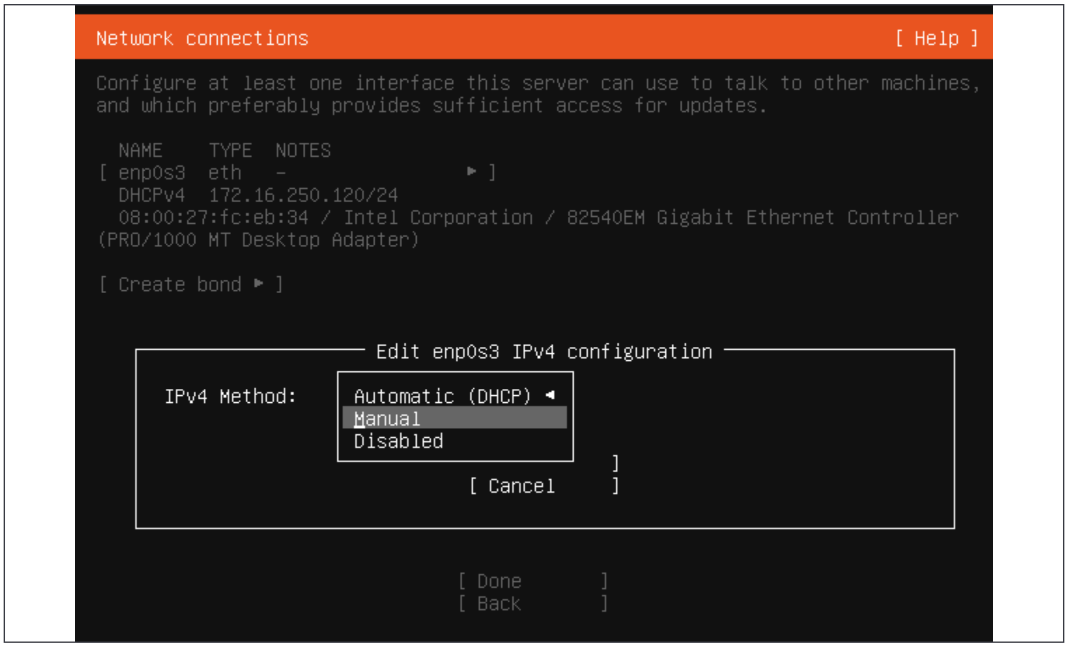
If you'd like to set up a static IP address, you can highlight the network card and press *Enter*, and then you can select **Edit IPv4** to begin the process of customizing your connection:



```
Network connections                                                          [ Help ]

Configure at least one interface this server can use to talk to other machines,
and which preferably provides sufficient access for updates.

  NAME    TYPE  NOTES
[ enp0s3  eth   -                       ▸ ◂ (close)
  DHCPv4  172.16.250.120/24                 Info            ▸
  08:00:27:fc:eb:34 / Intel Corpor          Edit IPv4       ▸ it Ethernet Controller
(PRO/1000 MT Desktop Adapter)               Edit IPv6       ▸
                                            Add a VLAN tag  ▸
[ Create bond ▸ ]

                                   [ Done          ]
                                   [ Back          ]
```

Figure 1.15: Selecting a network card and beginning the process of setting a custom IP address

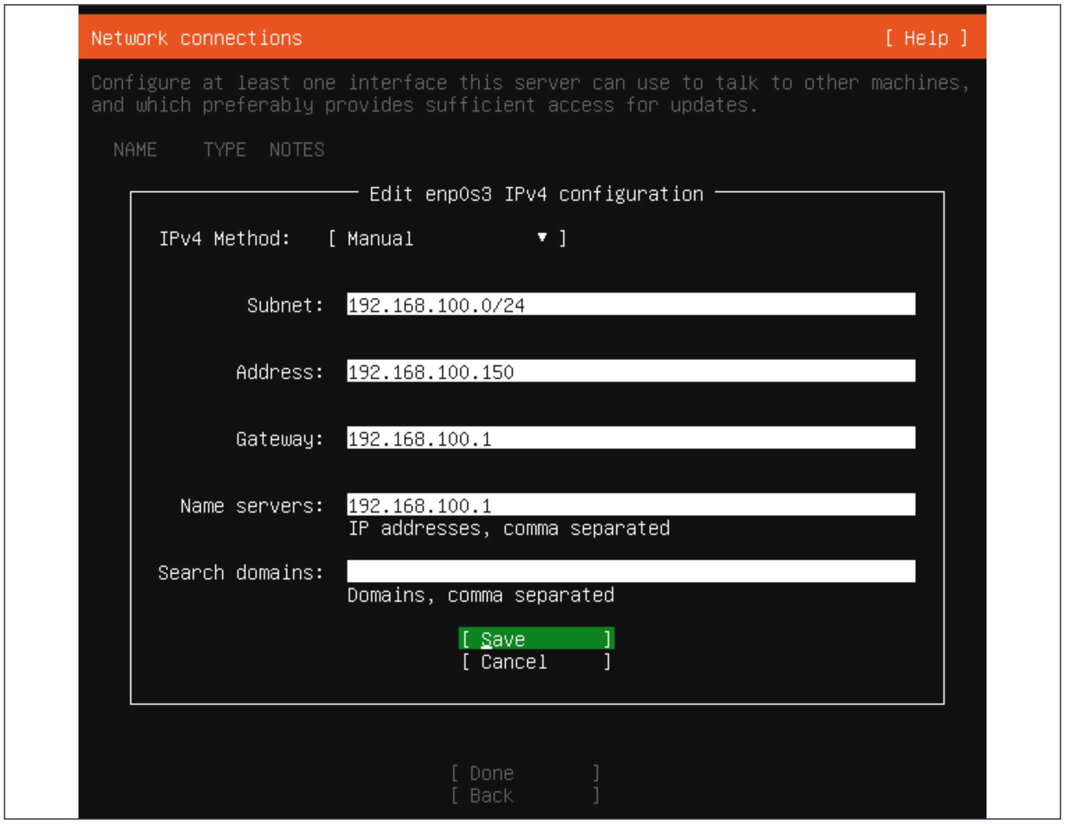
At the next screen, select **Manual** to move on to the next step:



```
Network connections                                                    [ Help ]

Configure at least one interface this server can use to talk to other machines,
and which preferably provides sufficient access for updates.

  NAME    TYPE  NOTES
[ enp0s3  eth   -                    ▸ ]
  DHCPv4  172.16.250.120/24
  08:00:27:fc:eb:34 / Intel Corporation / 82540EM Gigabit Ethernet Controller
(PRO/1000 MT Desktop Adapter)

[ Create bond ▸ ]

     ─────────────── Edit enp0s3 IPv4 configuration ───────────────
      IPv4 Method:    Automatic (DHCP) ◂
                      Manual
                      Disabled
                                                ]
                                 [ Cancel       ]

                                [ Done          ]
                                [ Back          ]
```

Figure 1.16: Static IP assignment (continued)

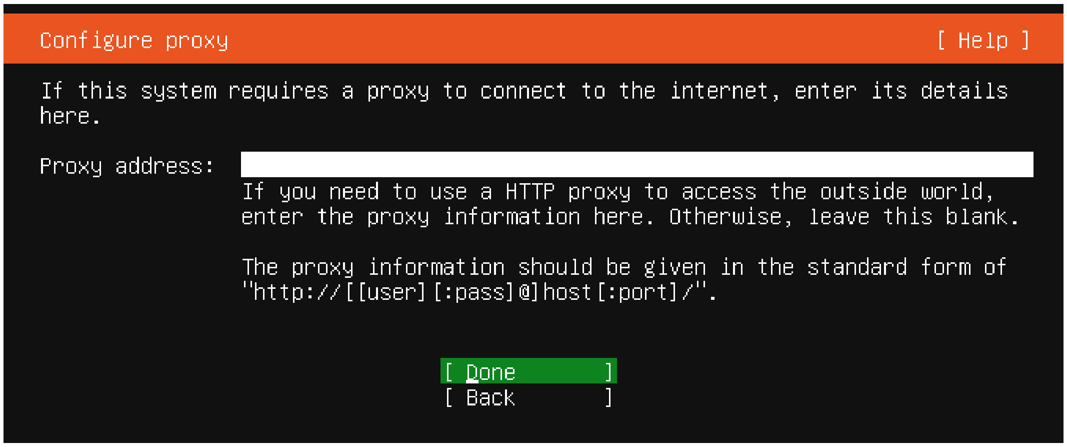
At the next screen, you'll be able to enter all the details specific to your network connection in order to set up your static IP assignment. When finished, select **Save**:



```
Network connections                                                  [ Help ]

Configure at least one interface this server can use to talk to other machines,
and which preferably provides sufficient access for updates.

  NAME    TYPE  NOTES

   ─────────────────── Edit enp0s3 IPv4 configuration ───────────────────

     IPv4 Method:   [ Manual           ▾ ]

          Subnet:  192.168.100.0/24

         Address:  192.168.100.150

         Gateway:  192.168.100.1

    Name servers:  192.168.100.1
                   IP addresses, comma separated

  Search domains:
                   Domains, comma separated

                             [ Save      ]
                             [ Cancel    ]

                            [ Done       ]
                            [ Back       ]
```

Figure 1.17: Entering network connection details

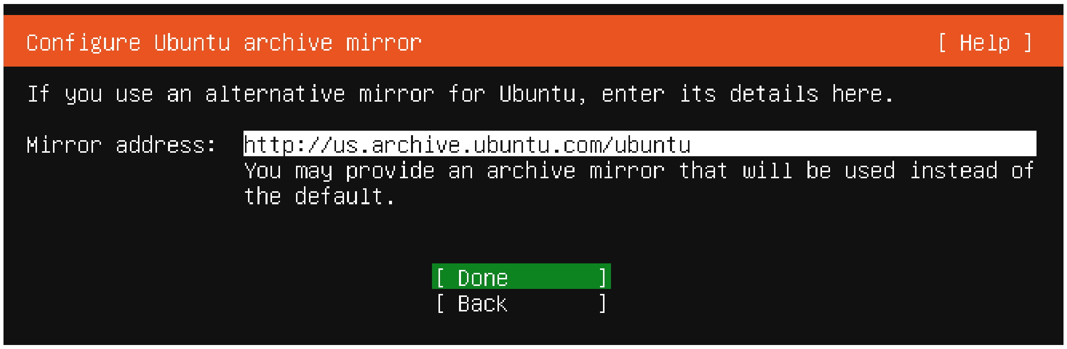
Next, the installer will give you an opportunity to enter your **Proxy address**, if you have one. Most readers will arrow down to done and press *Enter* to skip this, but if you do have an actual proxy, enter that here:



```
Configure proxy                                                    [ Help ]

If this system requires a proxy to connect to the internet, enter its details
here.

Proxy address:
                  If you need to use a HTTP proxy to access the outside world,
                  enter the proxy information here. Otherwise, leave this blank.

                  The proxy information should be given in the standard form of
                  "http://[[user][:pass]@]host[:port]/".


                              [ Done         ]
                              [ Back         ]
```

Figure 1.18: Ubuntu installer proxy prompt

Next, you'll be prompted to enter a **Mirror address** that will be used for Ubuntu's package manager (`apt`). Leave this URL as-is, and select **Done** to continue:



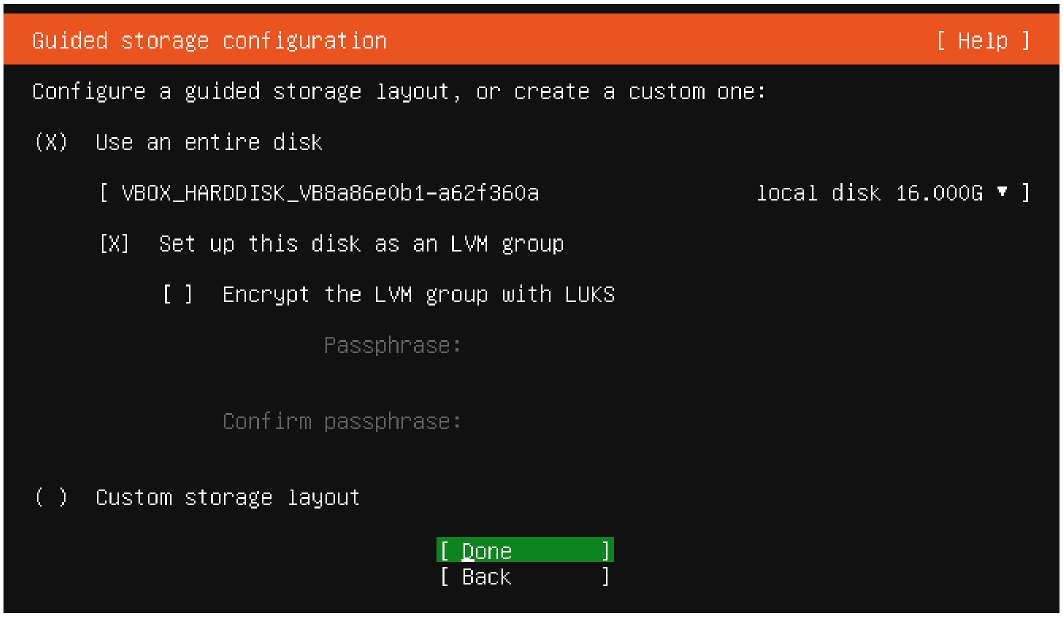
```
Configure Ubuntu archive mirror                                    [ Help ]

If you use an alternative mirror for Ubuntu, enter its details here.

Mirror address:   http://us.archive.ubuntu.com/ubuntu
                  You may provide an archive mirror that will be used instead of
                  the default.


                              [ Done         ]
                              [ Back         ]
```

Figure 1.19: Selecting a mirror for Ubuntu Server's package manager

The next section will give you a chance to configure your server's disk.

## Configuring the server's disk

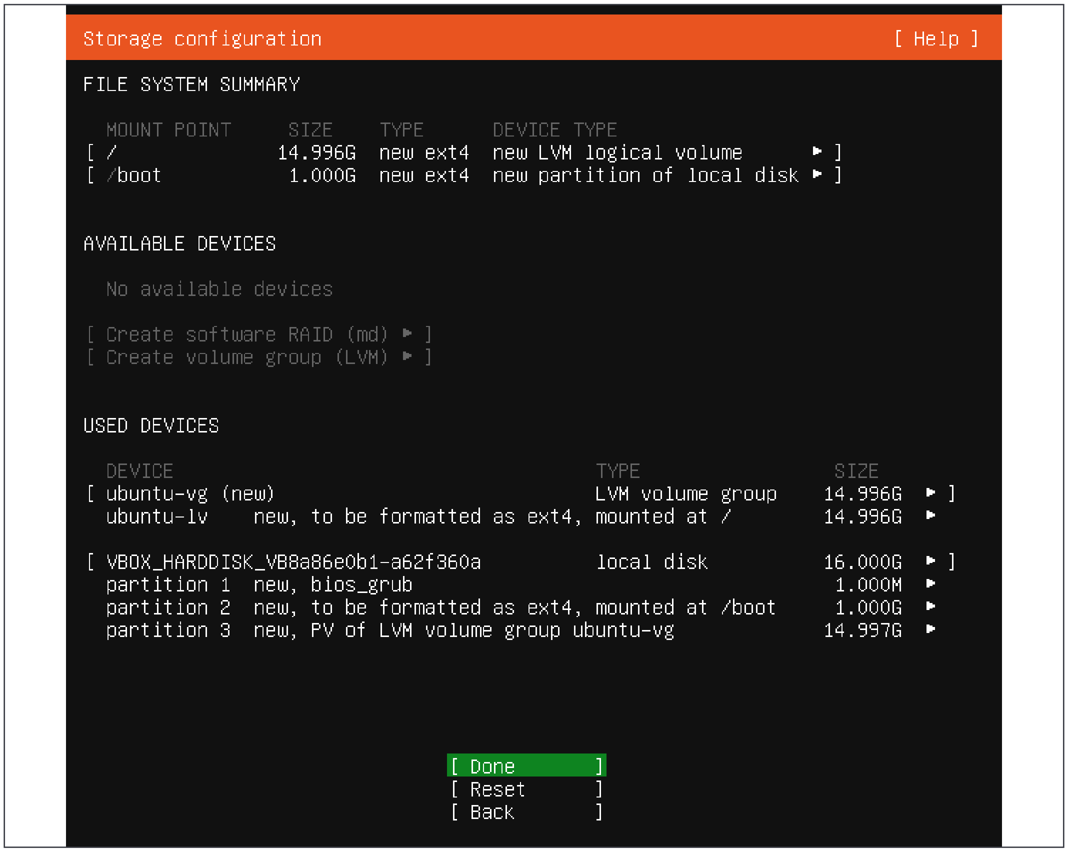
If you want to implement a custom partitioning scheme, as discussed earlier in the *Planning the partitioning layout* section, you can do that in the following screen. The **Custom storage layout** option will allow you to do just that. However, that is beyond the scope of this chapter, as we're just trying to get an installation off the ground, so choose **Use an entire disk** to continue:



```
Guided storage configuration                                              [ Help ]

Configure a guided storage layout, or create a custom one:

(X)  Use an entire disk

     [ VBOX_HARDDISK_VB8a86e0b1-a62f360a                  local disk 16.000G ▾ ]

     [X]  Set up this disk as an LVM group

          [ ]  Encrypt the LVM group with LUKS

                       Passphrase:


               Confirm passphrase:


( )  Custom storage layout

                                  [ Done        ]
                                  [ Back        ]
```

Figure 1.20: Disk setup screen of the Ubuntu Server installer

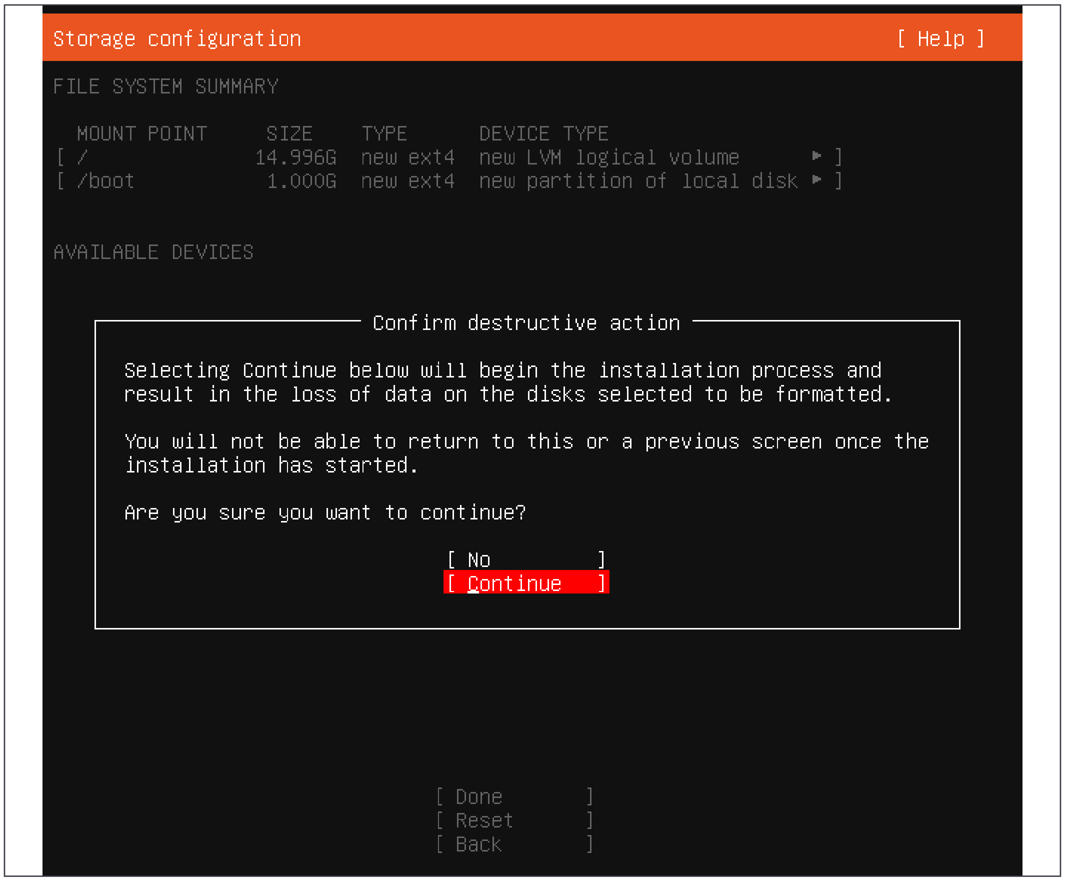
After selecting **Use an entire disk**, the next screen will show you a summary of the disk operations that will be performed. Select **Done** to continue with the default selections. Keep in mind that your disk will be erased as part of this process:



```
Storage configuration                                                  [ Help ]
FILE SYSTEM SUMMARY

  MOUNT POINT     SIZE    TYPE      DEVICE TYPE
[ /             14.996G  new ext4  new LVM logical volume      ▶ ]
[ /boot          1.000G  new ext4  new partition of local disk ▶ ]


AVAILABLE DEVICES

  No available devices

[ Create software RAID (md) ▶ ]
[ Create volume group (LVM) ▶ ]


USED DEVICES

  DEVICE                                     TYPE                SIZE
[ ubuntu-vg (new)                            LVM volume group    14.996G  ▶ ]
  ubuntu-lv    new, to be formatted as ext4, mounted at /        14.996G  ▶

[ VBOX_HARDDISK_VB8a86e0b1-a62f360a          local disk          16.000G  ▶ ]
  partition 1  new, bios_grub                                     1.000M  ▶
  partition 2  new, to be formatted as ext4, mounted at /boot     1.000G  ▶
  partition 3  new, PV of LVM volume group ubuntu-vg             14.997G  ▶



                                  [ Done        ]
                                  [ Reset       ]
                                  [ Back        ]
```

Figure 1.21: Showing a summary of disk operations during installation

To be doubly sure that you understand that the disk will be wiped and you're okay with that, the installer will show you another warning. Choose **Continue** at the following screen to move on:



```
Storage configuration                                                   [ Help ]
FILE SYSTEM SUMMARY

  MOUNT POINT     SIZE    TYPE      DEVICE TYPE
[ /             14.996G  new ext4  new LVM logical volume      ▸ ]
[ /boot          1.000G  new ext4  new partition of local disk ▸ ]


AVAILABLE DEVICES

     ───────────────────── Confirm destructive action ─────────────────────

       Selecting Continue below will begin the installation process and
       result in the loss of data on the disks selected to be formatted.

       You will not be able to return to this or a previous screen once the
       installation has started.

       Are you sure you want to continue?

                             [ No            ]
                             [ Continue      ]

                              [ Done         ]
                              [ Reset        ]
                              [ Back         ]
```

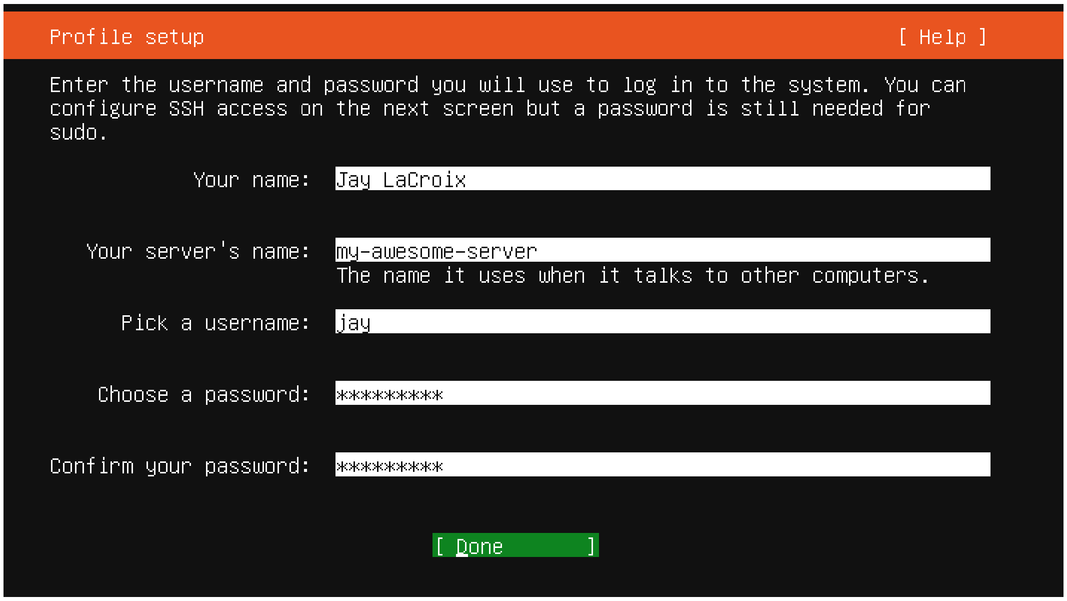
Figure 1.22: Final confirmation before writing changes to disk

Next, you'll then be brought to a screen where you can enter information related to the initial user account.

## Creating the initial user account

When we install Ubuntu Server, it will create an initial user account for us as part of the process. This user will be used for configuring and maintaining the server after installation. In this section of the installer, we'll be given an opportunity to name this user and create a password.

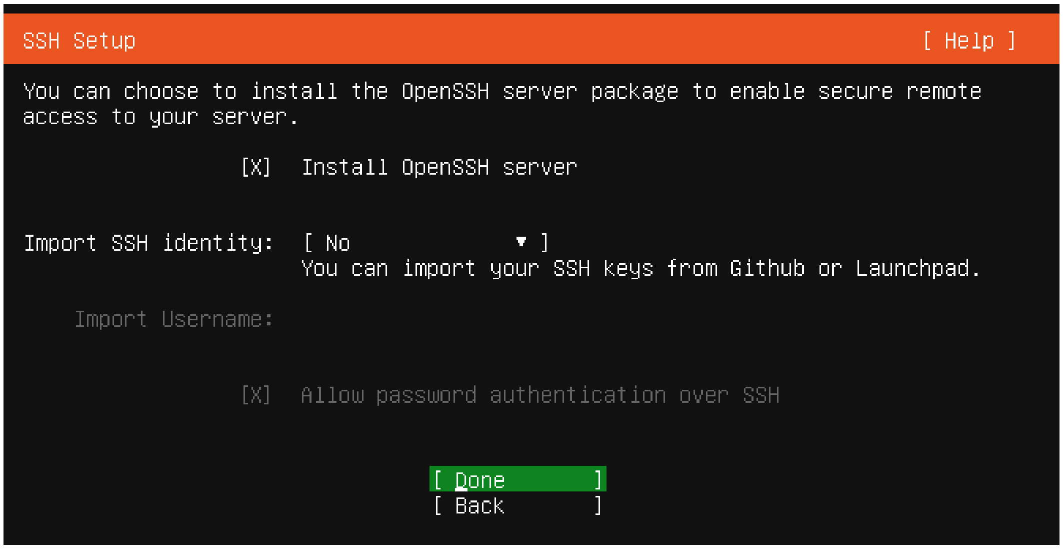
Fill out your information on this screen, as I have in the following screenshot. One important thing to keep in mind is that the user you create here will have administrative access. You can always create other users later on, but the user you create here will have special privileges. When finished, arrow down to **Done** and press *Enter*:



```
Enter the username and password you will use to log in to the system. You can
configure SSH access on the next screen but a password is still needed for
sudo.

              Your name: Jay LaCroix

       Your server's name: my-awesome-server
                           The name it uses when it talks to other computers.

         Pick a username: jay

       Choose a password: **********

   Confirm your password: **********
```

Figure 1.23: Entering details for the initial user account

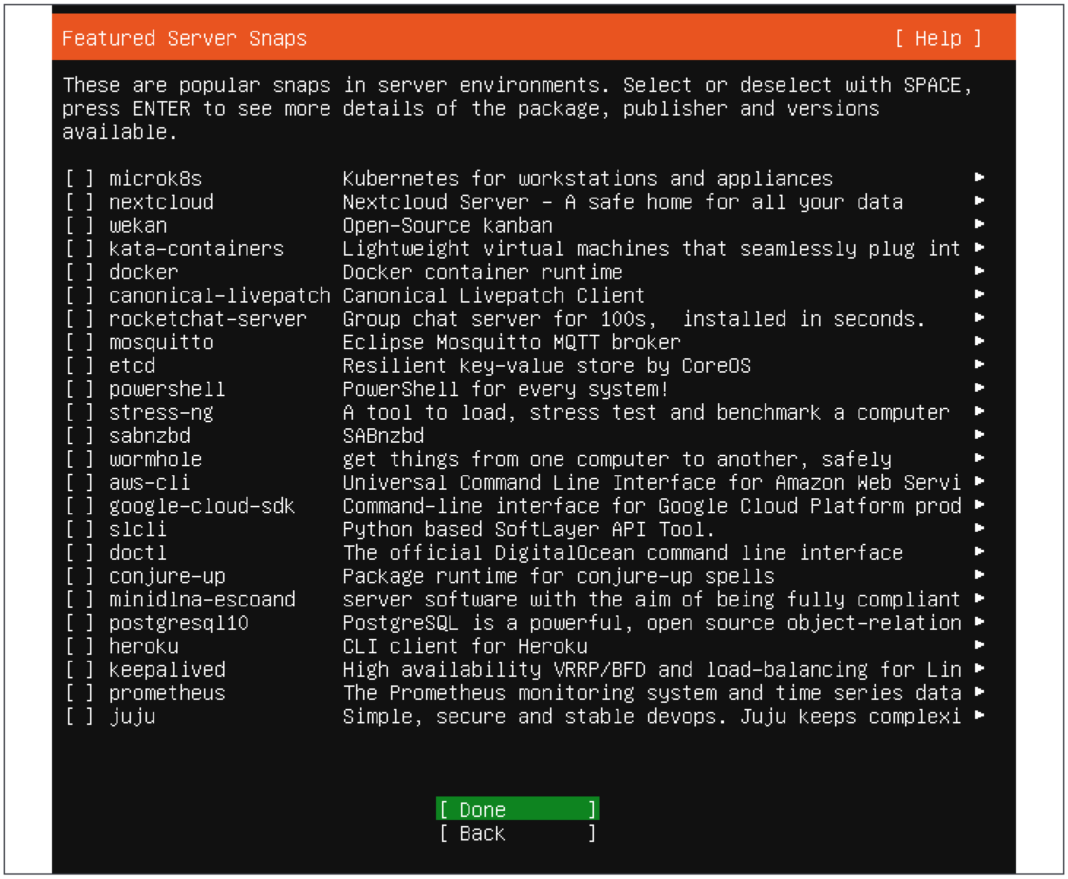
Next, you'll be given an option to install OpenSSH, which I recommend you do. This will allow you to easily connect to your server to perform remote management. OpenSSH is the gold standard for remote access in Linux and UNIX, so it will definitely be useful to have. Select **Done** to continue:



```
SSH Setup                                                                   [ Help ]

You can choose to install the OpenSSH server package to enable secure remote
access to your server.

                   [X]  Install OpenSSH server


Import SSH identity:  [ No                 ▾ ]
                      You can import your SSH keys from Github or Launchpad.

    Import Username:


                   [X]  Allow password authentication over SSH


                                    [ Done        ]
                                    [ Back        ]
```

Figure 1.24: Choosing to install OpenSSH during the installation of Ubuntu

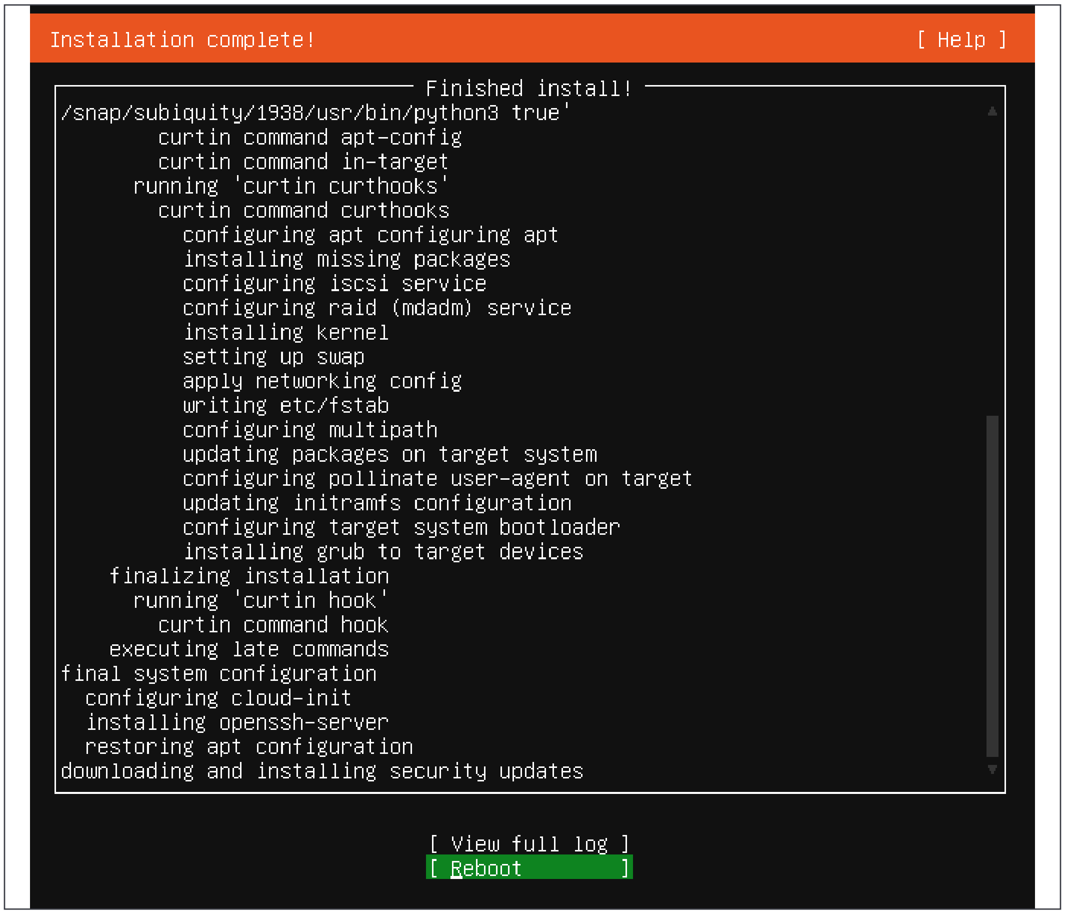
Although it's beyond the scope of this chapter, the new Ubuntu Server 20.04 installer allows you to import an SSH key from GitHub or Launchpad. Keep in mind that this feature exists, as that may be something you'll want to utilize later as it can simplify remote access. At the next screen, you'll see a large selection of additional applications that can be installed. We'll set up everything we need in this book manually as we get to each topic, so select **Done** to continue for now:



```
Featured Server Snaps                                              [ Help ]

These are popular snaps in server environments. Select or deselect with SPACE,
press ENTER to see more details of the package, publisher and versions
available.

[ ] microk8s            Kubernetes for workstations and appliances         ▸
[ ] nextcloud           Nextcloud Server - A safe home for all your data   ▸
[ ] wekan               Open-Source kanban                                 ▸
[ ] kata-containers     Lightweight virtual machines that seamlessly plug int ▸
[ ] docker              Docker container runtime                           ▸
[ ] canonical-livepatch Canonical Livepatch Client                         ▸
[ ] rocketchat-server   Group chat server for 100s,  installed in seconds. ▸
[ ] mosquitto           Eclipse Mosquitto MQTT broker                      ▸
[ ] etcd                Resilient key-value store by CoreOS                ▸
[ ] powershell          PowerShell for every system!                       ▸
[ ] stress-ng           A tool to load, stress test and benchmark a computer ▸
[ ] sabnzbd             SABnzbd                                            ▸
[ ] wormhole            get things from one computer to another, safely    ▸
[ ] aws-cli             Universal Command Line Interface for Amazon Web Servi ▸
[ ] google-cloud-sdk    Command-line interface for Google Cloud Platform prod ▸
[ ] slcli               Python based SoftLayer API Tool.                   ▸
[ ] doctl               The official DigitalOcean command line interface   ▸
[ ] conjure-up          Package runtime for conjure-up spells              ▸
[ ] minidlna-escoand    server software with the aim of being fully compliant ▸
[ ] postgresql10        PostgreSQL is a powerful, open source object-relation ▸
[ ] heroku              CLI client for Heroku                              ▸
[ ] keepalived          High availability VRRP/BFD and load-balancing for Lin ▸
[ ] prometheus          The Prometheus monitoring system and time series data ▸
[ ] juju                Simple, secure and stable devops. Juju keeps complexi ▸


                              [ Done       ]
                              [ Back       ]
```

Figure 1.25: Featured application selection

That's pretty much it—the installer will continue along on its own, as it has all the information it needs from you at this point. Once it's done, you'll see an option to **Reboot** at the bottom of the screen; go ahead and select that:



```
Installation complete!                                                    [ Help ]

                               Finished install!
/snap/subiquity/1938/usr/bin/python3 true'
        curtin command apt-config
        curtin command in-target
      running 'curtin curthooks'
        curtin command curthooks
          configuring apt configuring apt
          installing missing packages
          configuring iscsi service
          configuring raid (mdadm) service
          installing kernel
          setting up swap
          apply networking config
          writing etc/fstab
          configuring multipath
          updating packages on target system
          configuring pollinate user-agent on target
          updating initramfs configuration
          configuring target system bootloader
          installing grub to target devices
    finalizing installation
      running 'curtin hook'
        curtin command hook
    executing late commands
final system configuration
  configuring cloud-init
  installing openssh-server
  restoring apt configuration
downloading and installing security updates

                              [ View full log ]
                              [ Reboot        ]
```

Figure 1.26: Ubuntu Server setup is now complete and you are ready to restart into a fresh installation

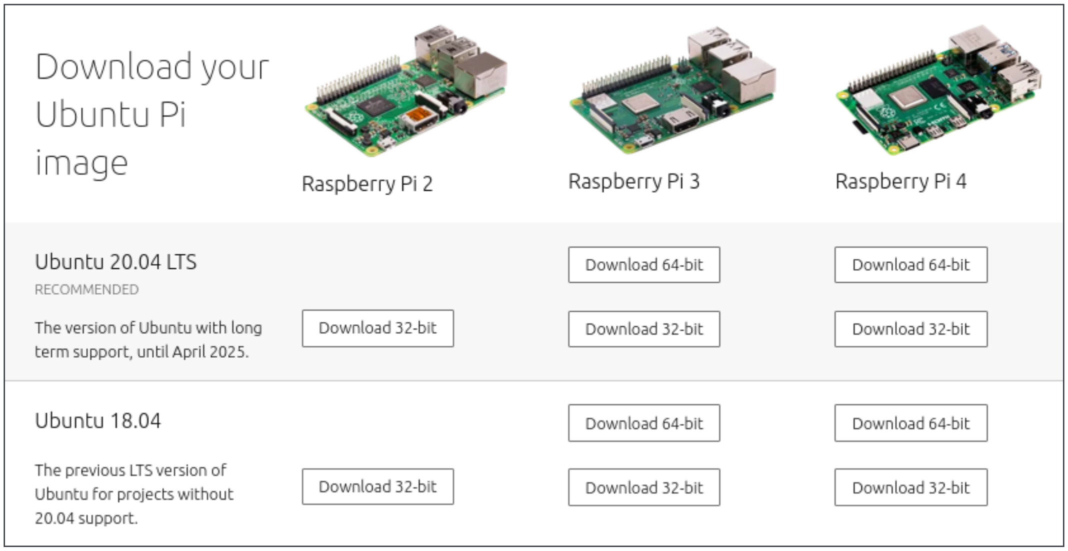
At this point, your server will reboot and then Ubuntu Server should start right up. Congratulations! You now have your own Ubuntu Server installation!

In addition to a physical device, Ubuntu Server can also be installed on a Raspberry Pi, and that's exactly what we'll take a look at in the next section.

## Installing Ubuntu on a Raspberry Pi

The Raspberry Pi platform has become quite a valuable asset in the industry, and a useful server platform. These tiny computers, now with four cores and up to 8 GB of RAM, are extremely power efficient and their performance is good enough that you can actually transform them into actual servers. In my lab, I have several Raspberry Pis on my network, each one responsible for performing specific tasks or functions. In many cases, you wouldn't even tell that they were devices with lower-powered hardware. Perhaps even more surprising is the fact that the Raspberry Pi 4 can outperform some lower-tier cloud instances, giving you a powerful server without the monthly cost if you don't need a high-end CPU. All this, in such a small form factor—these devices are smaller than the coaster under your coffee mug!

Ubuntu Server is available for Raspberry Pi models 2, 3, and 4. To get started, all you'll need to do is visit the official download page for the Raspberry Pi version of Ubuntu, here: `https://ubuntu.com/download/raspberry-pi`. This will take you to the following screen:



Download your Ubuntu Pi image

| | Raspberry Pi 2 | Raspberry Pi 3 | Raspberry Pi 4 |
|---|---|---|---|
| Ubuntu 20.04 LTS<br>RECOMMENDED<br>The version of Ubuntu with long term support, until April 2025. | Download 32-bit | Download 64-bit<br>Download 32-bit | Download 64-bit<br>Download 32-bit |
| Ubuntu 18.04<br>The previous LTS version of Ubuntu for projects without 20.04 support. | Download 32-bit | Download 64-bit<br>Download 32-bit | Download 64-bit<br>Download 32-bit |

Figure 1.27: The download section for Raspberry Pi on the official Ubuntu website

Once there, you'll see a listing of multiple versions for the Pi that are available for download. Refer to the preceding screenshot for an example of options that are available, but keep in mind that Ubuntu may redesign their web page at any time. So as a rule of thumb, download the version that matches the type of Raspberry Pi you have, and go with the 64-bit version if it's available (the Raspberry Pi 2 only has a 32-bit option available).

The download for the Pi versions will come in the form of a compressed image, which can be written directly to an SD card. To get the process going, open up Etcher, which we downloaded in an earlier step. Once Etcher is open, click **Flash from file**:

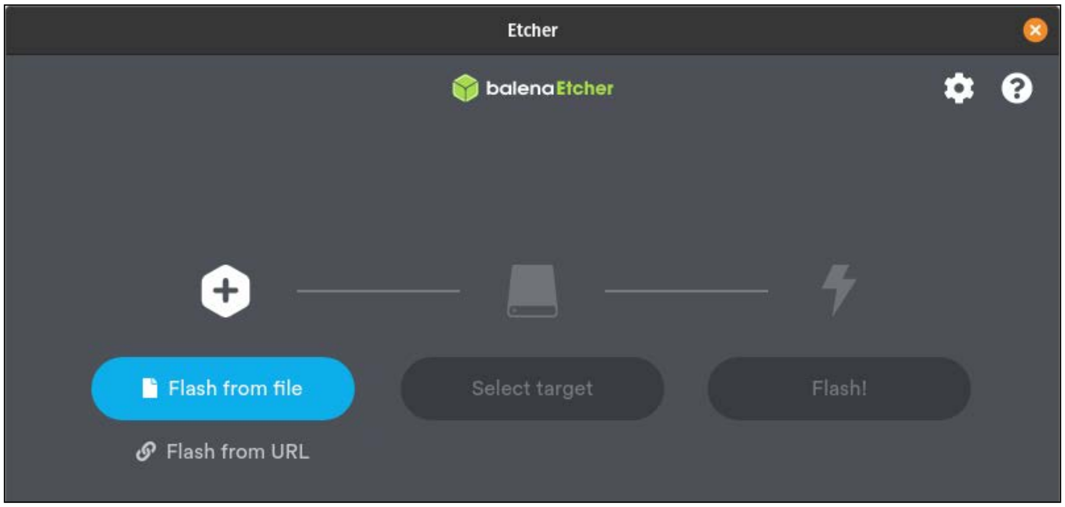
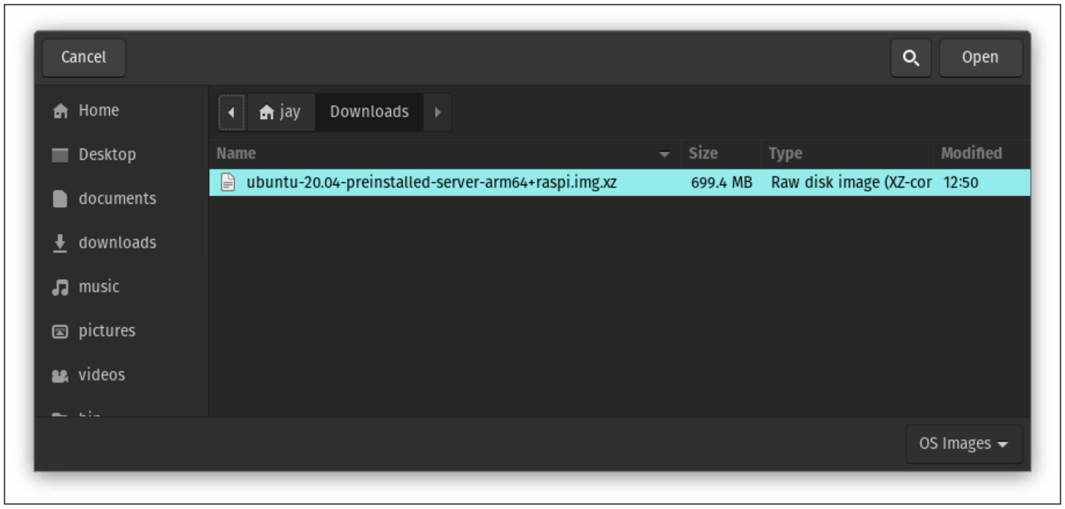


Figure 1.28: Etcher, ready for action

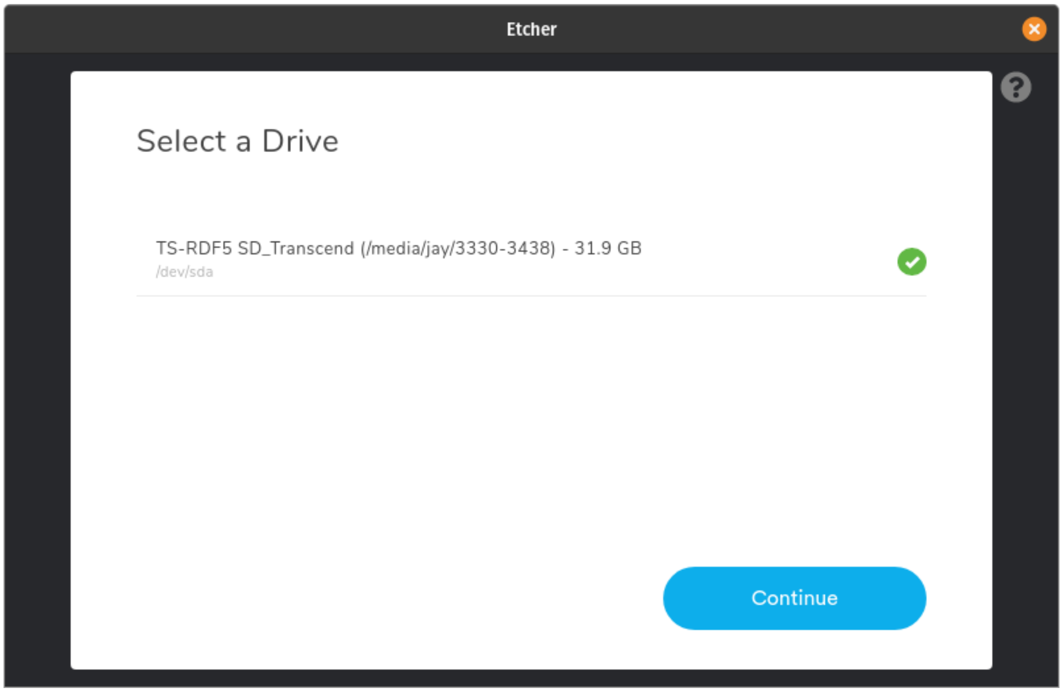
Now, choose the file that was downloaded earlier and select **Open**:



| Name | Size | Type | Modified |
|---|---|---|---|
| ubuntu-20.04-preinstalled-server-arm64+raspi.img.xz | 699.4 MB | Raw disk image (XZ-cor | 12:50 |

Figure 1.29: Selecting the file to use to write to the SD card

Next, go ahead and insert an SD card into your computer that will become dedicated to Ubuntu Server. Etcher will completely erase the SD card, so make sure you don't care about any data that may be on it. If your computer does not have an SD card slot, you can use a USB card reader. If the proper device isn't selected, you can click **Select target** to choose the SD card to use, as shown in *Figure 1.28*. Then click **Continue**:



TS-RDF5 SD_Transcend (/media/jay/3330-3438) - 31.9 GB
/dev/sda

Figure 1.30: Choosing an SD card to use with Etcher for the Raspberry Pi

Once you've selected both the image file to use and the target SD card, you're ready to proceed. Click **Flash!** to begin the process of writing the image:

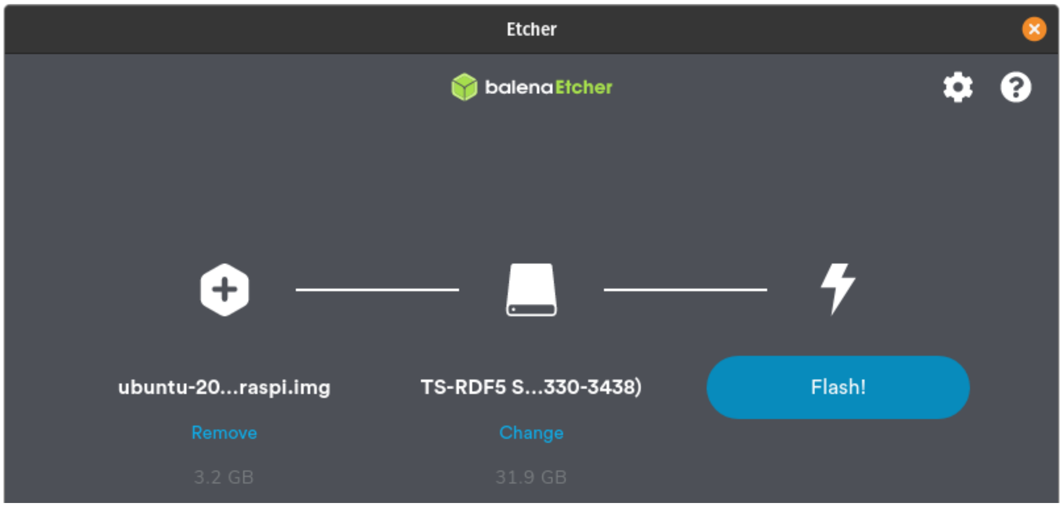


Figure 1.31: Etcher, ready to start flashing the SD card

Now, Etcher will prepare the target with the source image file. The process can take anywhere from 5 to 15 minutes depending on the speed of your hardware:

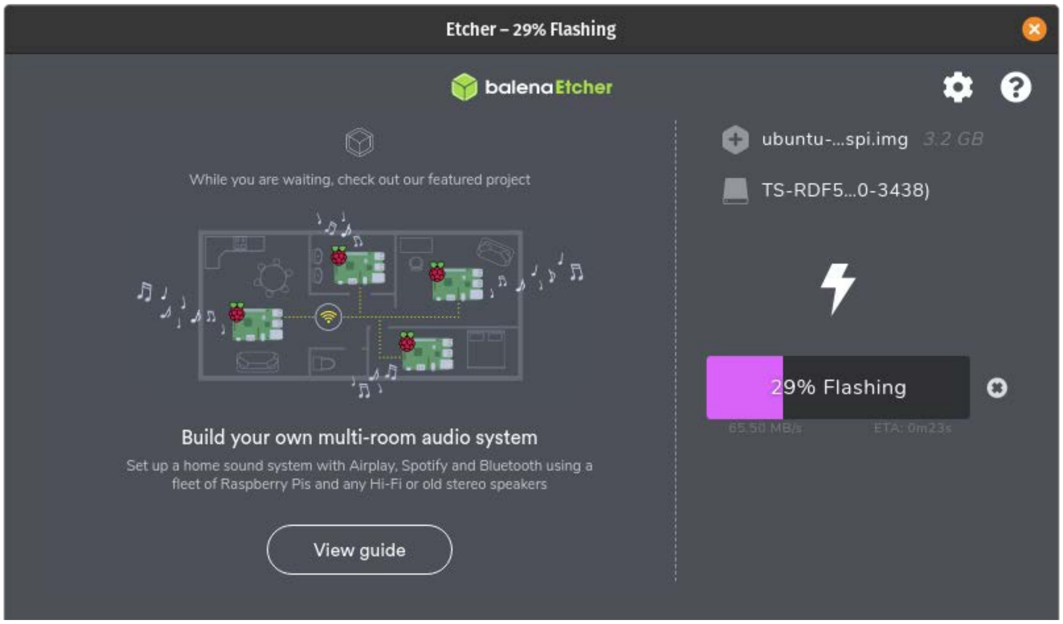

Etcher – 29% Flashing
balenaEtcher
While you are waiting, check out our featured project
Build your own multi-room audio system
Set up a home sound system with Airplay, Spotify and Bluetooth using a fleet of Raspberry Pis and any Hi-Fi or old stereo speakers
View guide
ubuntu-...spi.img 3.2 GB
TS-RDF5...0-3438)
29% Flashing
65.50 MB/s
ETA: 0m23s

Figure 1.32: Etcher in progress

Once Etcher is finished writing to the SD card, you can eject it from your computer and insert it into the Pi, which can be accessed remotely via SSH. When it finishes booting for the first time, you can log in and begin using it as you would any other Ubuntu Server installation (the username and password both default to `ubuntu`). From there, you can create a new user for yourself, install applications, and experiment.

## Summary

In this chapter, we covered a couple of different installation processes in great detail. As with most Linux distributions, Ubuntu Server is scalable and able to be installed on a variety of server types. Physical devices, VMs, and even the Raspberry Pi have versions of Ubuntu available. The process of installation was covered step by step, and you should now have an Ubuntu Server instance of your own to configure as you wish. Also in this chapter, we covered determining the role of your server, the process of creating bootable media, and a walk-through of setting up Ubuntu Server on a Raspberry Pi. You're now on your way to Ubuntu Server mastery.

In the next chapter, we'll show you how to manage users. You'll be able to create them, delete them, change them, and even manage password expiration and more. We will also cover permissions so that you can determine what your users are allowed to do on your server. See you there!

## Further Reading

- Install Ubuntu Server (Video Tutorial) from LearnLinuxTV: `https://learnlinux.link/install_ubuntu_server`

# 2
# Managing Users and Permissions

In the previous chapter, we set up our very own Ubuntu Server installation, and we can now learn how to maintain it, starting with a look at managing who is able to use our server.

As an administrator of Ubuntu servers, users can be your greatest asset and also your biggest headache. During your career, you'll add countless new users, manage their passwords, remove their accounts when they leave the company, and grant or remove access to resources across the filesystem. Even on servers on which you're the only user, you'll still find yourself managing user accounts, since even system processes run as users. To be successful at managing Linux servers, you'll also need to know how to manage permissions, create password policies, and limit who can execute administrative commands on the machine. In this chapter, we'll work through these concepts so that you'll have a clear idea of how to manage users and their resources.

In particular, we will cover:

- Understanding users and groups
- Understanding when to use `root`
- Creating and removing users
- Understanding the `/etc/passwd` and `/etc/shadow` files
- Distributing default configuration files with `/etc/skel`
- Switching users

- Managing groups
- Managing passwords and password policies
- Configuring administrator access with `sudo`
- Setting permissions on files and directories

In the first section, we have a quick discussion of the nature of managing users.

## Understanding users and groups

When it comes to a server, users are very important—without users to serve, then there's no real need for a server in the first place. The subject of user management itself within the world of IT is in and of itself quite vast. Entire books have been written on individual methods of authentication, and entire technologies (such as **Lightweight Directory Access Protocol**, or **LDAP**) exist around it. In this chapter, we'll look at managing users that exist locally to our server, and the groups that help define what they are able to do.

Since Ubuntu Server is a distribution of Linux, it adopts the Unix-style of managing user accounts, groups, and permissions. Although our focus is on Ubuntu, many of the same commands around user management that you'll learn in this chapter will apply to other platforms as well. There are commands that allow you to add, remove, and change users, as well as commands that allow you to alter permissions.

Users in the context of a server refer to who (or what) is able to use the server. For example, you may have an accountant named Susan, or an IT administrator named Haneef, who both need to access the server. Perhaps Susan only needs access to a file share directory for accounting-related files, and Haneef might have more access to the server as a system administrator. The user accounts we create on our server will represent the actual people that will use it.

Groups allow us to segregate access to be specific to a role. As we'll learn later, files and directories have user and group assignments. When combined with permissions, we'll be able to manage what our users are able to do with our server.

Users aren't always people, though. We also have system users on our server that applications and running processes might use for background or automated tasks. An example of this might be a backup job, and you may have a backup user that runs a task in the background to facilitate some sort of file copy task that copies important files to another place. You don't have to worry about system-related users for now, just know that they exist. You'll see more examples of this as we go through the book.

More advanced organizations may have a central login server, such as **Active Directory** (**AD**) or standard LDAP. There are also others aside from those, as well. In this book, we won't cover those technologies, but just keep in mind that central authentication servers are a possibility for your organization, should you choose to explore them.

The most powerful user of all, though, is `root`. This special user gives us the most control, but as you'll see in the next section, that comes with risks.

## Understanding when to use root

In the last chapter, we set up our very own Ubuntu Server installation. During the installation process, we were instructed to create a user account to act as a system administrator. So, at this point, we should have at least two users on our server. We have the aforementioned administrative user, as well as `root`. We can certainly create additional user accounts with varying levels of access (and we will do so in this chapter), but before we get to that, some discussion is in order regarding the administrator account you created, as well as the `root` user that was created for you.

The `root` user account exists on all Linux distributions and is the most powerful user account on the planet. The `root` user account can be used to do anything within your server, and I do mean *anything*. Want to create files and directories virtually anywhere on the filesystem? Want to install software? These processes are easily performed with `root`. The `root` account can even be used to destroy your entire installation with one typo or ill-conceived command: if you instruct `root` to delete all the files on your entire hard disk, it won't hesitate to do so. It's always assumed on a Linux system that if you are using `root`, you are doing so because you know what you are doing. So, there's often not so much as a confirmation prompt while executing any command as `root`. It will simply do as instructed, for better or worse.

It's for this reason that every Linux distribution I've ever used states, or at least highly recommends, that you should create a standard user during the installation process. It's generally recommended in the Linux community for an administrator to have their own account and then switch to `root` whenever a task comes up that requires `root` privileges to complete. This approach is less likely to destroy your server with an accidental typo or bad command while you're logged in as `root`. Some administrators will strictly use `root` at all times without any issue, but again, it's recommended to use `root` only when you have to.

Most distributions ask you to create a `root` password during installation in order to protect that account. Even Debian (on which Ubuntu is based) has you set a `root` password during installation. Ubuntu just decides to do things a little bit differently. The reason for this is because, unlike many other distributions, Ubuntu defaults to locking out the `root` account altogether. There's nothing stopping you from enabling `root`, or switching to the `root` user after you log in. Being disabled by default just means the `root` account isn't as easily accessible as it normally would be. I'll cover how to enable this account later in this chapter, should you feel the need to do so.

An exception to this rule is that some VPS providers, such as DigitalOcean, will enable the `root` account even on their Ubuntu servers. Typically, the `root` password will be randomly generated and emailed to you. However, you should still create a user for yourself with administrative access regardless.

Instead of using `root` outright, Ubuntu (as well as its server version) recommends the use of `sudo`.

## Using sudo to run privileged commands

I'll go over how to manage `sudo` later on in this chapter, but for now, just keep in mind that the purpose of `sudo` is to enable you to use your user account to do things that normally only `root` would be able to do. For example, as a normal user, you cannot issue a command such as the following to install a software package:

```
apt install tmux
```

Instead, you'll receive an error:

```
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
Permission denied)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-
frontend), are you root?
```

But if you prefix the command with `sudo` (assuming your user account has access to it), the command will work just fine:

```
sudo apt install tmux
```

When you use sudo, you'll be asked for your user's password for confirmation, and then the command will execute. Subsequent commands prefixed with sudo may not prompt for your password, as it will cache your password for a short period of time until it times out or the terminal is closed. Understanding this should clarify the usefulness of the user account you created during installation. I referred to this user as an administrative account earlier, but it's really just a user account that is able to utilize sudo.

> Ubuntu Server automatically gives the first user account you create during installation access to sudo.

The intent is that you'll use that account to administer the system, rather than root. When you create additional user accounts, they will not have access to sudo by default, unless you explicitly grant it to them.

## Creating and removing users

Creating users in Ubuntu can be done with one of two commands: adduser and useradd. This can be a little confusing at first, because both of these commands do the same thing (in different ways) and are named very similarly. I'll go over the useradd command first and then I'll explain how adduser differs. You may even prefer the latter, but we'll get to that in a moment.

### Using useradd

First, here's an example of the useradd command in action:

```
sudo useradd -d /home/jdoe -m jdoe
```

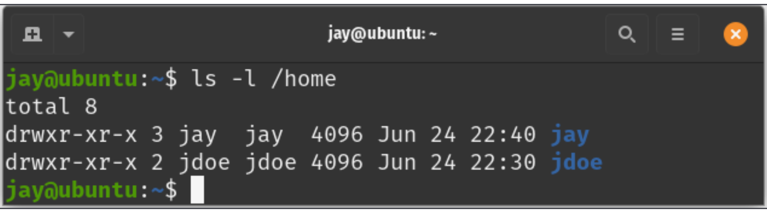
With this command, I created a user named jdoe. With the -d option, I'm clarifying that I would like a home directory created for this user, and following that, I called out /home/jdoe as the user's home directory. The -m flag tells the system that I would like the home directory to be created during the process; otherwise, I would've had to create the directory myself. Finally, I called out the username for my new user (in this case, jdoe).

As we go along in this book, there will be commands that require `root` privileges in order to execute. The preceding command was an example of this. For commands that require such permissions, I'll prefix the commands with `sudo`. When you see these, it just means that `root` privileges are required to run the command. For these, you can also log in as `root` (if `root` is enabled) or switch to `root` to execute these commands as well. However, as I mentioned before, using `sudo` instead of using the `root` account is strongly encouraged.

Now, list the storage of `/home` using the following command:

```
ls -l /home
```

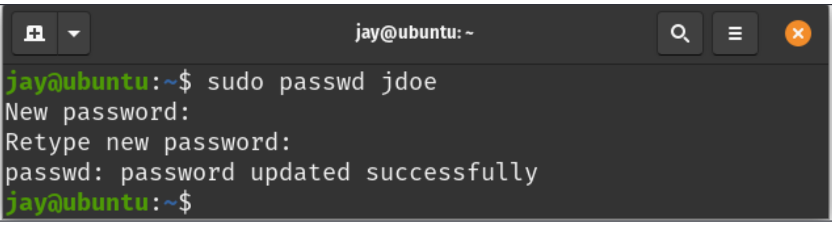
You should see a folder listed there for our new user:



```
jay@ubuntu:~$ ls -l /home
total 8
drwxr-xr-x 3 jay  jay  4096 Jun 24 22:40 jay
drwxr-xr-x 2 jdoe jdoe 4096 Jun 24 22:30 jdoe
jay@ubuntu:~$
```

Figure 2.1: Listing the contents of /home after our first user was created

What about creating our user's password? We may have been asked for our current user's password due to using `sudo`, but we weren't asked for a password for the new user. To create a password for the user, we can use the `passwd` command. The `passwd` command defaults to allowing you to change the password for the user you're currently logged in as, but it also allows you to set a password for any other user if you run it as `root` or with `sudo`. If you enter `passwd` by itself, the command will first ask you for your current password, then your new password, and then it will ask you to confirm your new password again. If you prefix the command with `sudo` and then specify a different user account, you can set the password for any user you wish. An example of the output of this process is as follows:



```
jay@ubuntu:~$ sudo passwd jdoe
New password:
Retype new password:
passwd: password updated successfully
jay@ubuntu:~$
```

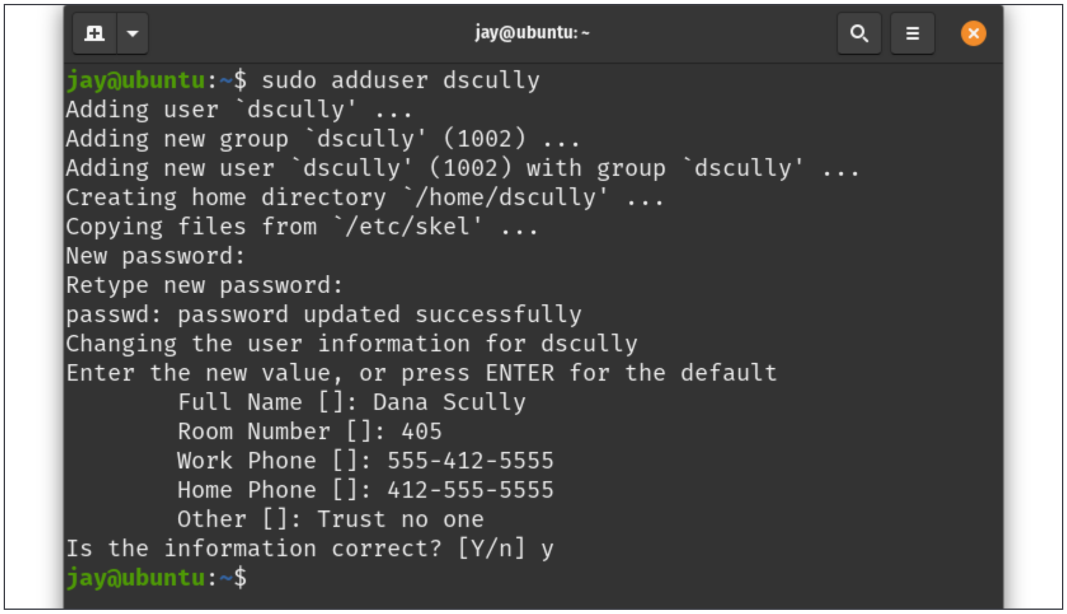
Figure 2.2: Changing the password of a user

As you can see in the previous screenshot, you won't see any asterisks or any kind of output when you type a password using the `passwd` command. This is normal. Although you won't see any visual indication of input, your input is being recognized.

Now we have a new user and we were able to set a password for that user. The `jdoe` user will now be able to access the system with the password we've chosen. This user won't have access to `sudo` by default, but we'll cover how to change this later on in the chapter.

## Using adduser

Earlier, I mentioned the `adduser` command as another way of creating a user. The difference (and convenience) of this command should become apparent immediately once you've used it. Go ahead and give it a try; execute `adduser` along with a username for a user you wish to create. An example run of this process is as follows:



```
jay@ubuntu:~$ sudo adduser dscully
Adding user `dscully' ...
Adding new group `dscully' (1002) ...
Adding new user `dscully' (1002) with group `dscully' ...
Creating home directory `/home/dscully' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for dscully
Enter the new value, or press ENTER for the default
        Full Name []: Dana Scully
        Room Number []: 405
        Work Phone []: 555-412-5555
        Home Phone []: 412-555-5555
        Other []: Trust no one
Is the information correct? [Y/n] y
jay@ubuntu:~$
```

Figure 2.3: Creating a user with the adduser command

In the preceding process, I executed `sudo adduser dscully` (commands that modify users require `sudo` or `root`) and then I was asked a series of questions regarding how I wanted the user to be created. I was asked for the password (twice), `Full Name`, `Room Number`, `Work Phone`, and `Home Phone`. In the `Other` field, I entered the comment `Trust no one`, which is a great mindset to adopt while managing users. The latter prompts prior to the final confirmation were all optional: I didn't have to enter `Full Name`, `Room Number`, and so on. I could've pressed *Enter* to skip those prompts if I wanted to. The only thing that's really required is the username and the password.

From the output, we can see that the `adduser` command performed quite a bit of work for us. The command defaulted to using `/home/dscully` as the home directory for the user, the account was given the next available **User ID** (**UID**) and **Group ID** (**GID**) of `1002`, and it also copied files from `/etc/skel` into our new user's `home` directory. In fact, both the `adduser` and `useradd` commands copy files from `/etc/skel`, but `adduser` is more verbose regarding the actions it performs.

Don't worry if you don't understand `UID`, `GID`, and `/etc/skel` yet. We'll work through those concepts soon.

In a nutshell, the `adduser` command is much more convenient in the sense that it prompts you for various options while it creates the user without requiring that you memorize command-line options. It also gives you detailed information about what it has done. At this point, you may be wondering why someone would want to use `useradd` at all, considering how much more convenient `adduser` seems to be. Unfortunately, `adduser` is not available on all distributions of Linux. It's best to familiarize yourself with `useradd` in case you find yourself on a Linux system that's not Ubuntu.

It may be interesting for you to see what exactly the `adduser` command is. It's not even a binary program—it's a **shell script**. A shell script is simply a text file that can be executed as a program. You don't have to worry too much about scripting now, as we will cover it in *Chapter 6*, *Boosting Your Command-Line Efficiency*. In the case of `adduser`, it's a script written in **Perl**, which is a programming language that is sometimes used for administrative tasks. Since it's not binary, you can even open it in a text editor in order to view all the code that it executes behind the scenes. However, make sure you don't open the file in a text editor with `root` privileges, to ensure that you don't accidentally save changes to the file and break the script. The following command will open `adduser` in a text editor on an Ubuntu Server system:

```
nano /usr/sbin/adduser
```

Use your up/down arrows as well as the *Page Up* and *Page Down* keys to scroll through the file. When you're finished, press *Ctrl* + *x* on your keyboard to exit the text editor.

Those of you with keen eyes will likely notice that the `adduser` script is calling `useradd` to perform its actual work. So either way, you're either directly or indirectly using `useradd`.

Now that we know how to create users, it will be useful to understand how to remove them as well.

## Removing users

Removing access is very important when a user no longer needs to access a system, as unmanaged accounts often become a security risk. To remove a user account, we'll use the `userdel` command.

Before removing an account, though, there is one very important question you should ask yourself. Will you still need access to the user's files? Most companies have retention policies in place that detail what should happen to a user's data when he or she leaves the organization. Sometimes, these files are copied into an archive for long-term storage. Often, a manager, coworker, or a new hire will need access to the former user's files, perhaps to continue working on a project where they left off. It's important to understand this policy ahead of managing users. If you don't have a policy in place that outlines retention requirements for files when users resign, you should probably work with your management and create one.

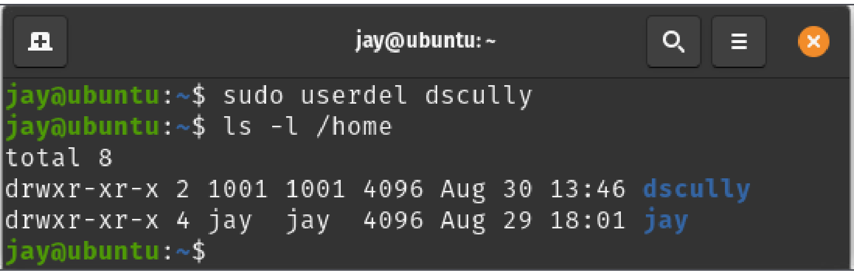
By default, the `userdel` command does not remove the contents of the user's `home` directory. Here, we use the following command to remove `dscully` from the system:

```
sudo userdel dscully
```

We can see that the files for the `dscully` user still exist by entering the following command:

```
ls -l /home
```

The preceding commands will result in the following outputs:



```
jay@ubuntu:~$ sudo userdel dscully
jay@ubuntu:~$ ls -l /home
total 8
drwxr-xr-x 2 1001 1001 4096 Aug 30 13:46 dscully
drwxr-xr-x 4 jay  jay  4096 Aug 29 18:01 jay
jay@ubuntu:~$
```

Figure 2.4: The home directory for the user dscully still exists, even though we removed the user

With the `/home` directory for `dscully` still existing, we're able to move the contents of this directory anywhere we would like to. If we had a directory called `/store/file_archive`, for example, we could easily move the files there:

```
sudo mv /home/dscully /store/file_archive
```

Of course, it's up to you to create the directory where your long-term storage will ultimately be, but you get the idea.

If you weren't already aware, you can create a new directory with the `mkdir` command. You can create a directory within any other directory that your logged-in user has access to. The following command will create the `file_archive` directory I mentioned in the preceding example:

```
sudo mkdir -p /store/file_archive
```

The `-p` flag simply creates the parent directory if it didn't already exist.

If you do actually want to remove a user's home directory at the same time that you remove an account, just add the `-r` option. This will eliminate the user and their data in one shot:

```
sudo userdel -r dscully
```

To remove the `/home` directory for the user after the account was already removed (if you didn't use the `-r` parameter the first time), use the `rm -r` command to get rid of it, as you would any other directory:

```
sudo rm -r /home/dscully
```

It probably goes without saying, but the `rm` command can be extremely dangerous. If you're logged in as `root` or using `sudo` while using `rm`, you can easily destroy your entire installed system if you're not careful. *DO NOT run this command*, but as a hypothetical example, the following command (while seemingly innocent at first glance) will likely completely destroy your entire filesystem:

```
sudo rm -r / home/dscully
```

Notice the typo: I accidentally typed a space after the first forward slash. I literally accidentally told my system to remove the contents of the entire filesystem. If that command was executed, the server probably wouldn't even boot the next time we attempted to start it. All user and program data would be wiped out. If there was ever any single reason for us to be protective over the `root` account, the `rm` command is certainly it!

At this point, we understand how to add and remove users. In the next section, we'll look deeper into passwords.

## Understanding the /etc/passwd and /etc/shadow files

Now that we know how to create (and delete) user accounts on our server, we are well on our way to being able to manage our users. But where exactly is this information stored? We know that users store their personal files in `/home`, but is there some kind of database somewhere that keeps track of which user accounts are on our system? Actually, user account information is stored in two special text files:

- `/etc/passwd`
- `/etc/shadow`

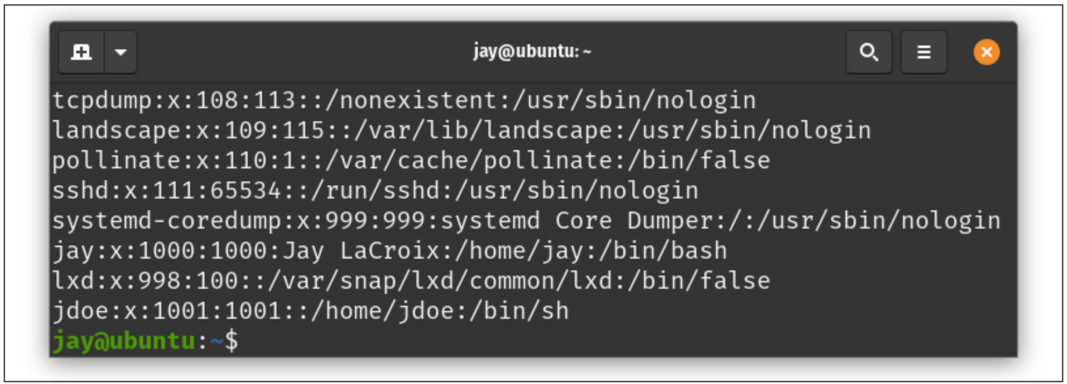
You can display the contents of each of those two files with the following commands. Note that any user can look at the contents of `/etc/passwd`, while only `root` has access to `/etc/shadow`:

```
cat /etc/passwd
sudo cat /etc/shadow
```

Go ahead and take a look at these two files (just don't make any changes), and I will help you understand them.

## Using /etc/passwd

First, let's go over the `/etc/passwd` file. What follows is some example output from this file on my test server. For brevity, I have limited the output to the last eight lines:



```
tcpdump:x:108:113::/nonexistent:/usr/sbin/nologin
landscape:x:109:115::/var/lib/landscape:/usr/sbin/nologin
pollinate:x:110:1::/var/cache/pollinate:/bin/false
sshd:x:111:65534::/run/sshd:/usr/sbin/nologin
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
jay:x:1000:1000:Jay LaCroix:/home/jay:/bin/bash
lxd:x:998:100::/var/snap/lxd/common/lxd:/bin/false
jdoe:x:1001:1001::/home/jdoe:/bin/sh
jay@ubuntu:~$
```

Figure 2.5: Example /etc/passwd file

Each line within this file corresponds to a user account on the system. Entries are split into columns, separated by a colon (`:`). The username is in the first column, so you can see that I've created users `jay` and `jdoe`. The next column on each is simply an `x`. I'll go over what that means a bit later. For now, let's skip to the third and fourth columns, which reference the UID and GID respectively. On a Linux system, user accounts and groups are actually referenced by their IDs. While it's easier for you and I to manage users by their names, usernames and group names are nothing more than a label placed on the UID and GID in order to help us identify them easier.

For example, it may be frustrating to try to remember that `jdoe` is UID `1001` on our server each time we want to manage this account. Managing it by referring to the account as `jdoe` is easier for humans, since we don't remember numbers as well as we do names. But to Linux, each time we reference user `jdoe`, we're actually just referencing UID `1001`. When a user is created, the system (by default) automatically assigns the next available UID to the account. If you manage multiple Ubuntu servers, note that the UIDs will not match from one system to another, so keep in mind that UIDs don't synchronize between installations.

In my case (as shown in *Figure 2.5*), the UID of each user is the same as their GID. This is just a coincidence on my system and it isn't always that way in practice. While I'll discuss creating groups later in this chapter, understand that creating groups works in a similar way to creating users, in the sense that the group is assigned the next available GID in much the same way as new user accounts are assigned the next available UID. When you create a user, the user's primary group is the same as their username (unless you request otherwise). For example, when I created `jdoe`, the system also automatically created a `jdoe` group as well.

This is what you're actually seeing here—the UID for the user, as well as the GID for the user's primary group. Again, we'll get to groups in more detail later.

You probably also noticed that the `/etc/passwd` file on your system contains entries for many more users than the ones we've created ourselves. This is perfectly normal, as Linux uses user accounts for various processes and services that run in the background. You'll likely never interact with the default accounts at all, though you may someday create your own system user for a process to run as. For example, perhaps you'll create a data processor account for an automated data-processing script to run under.

Anyway, back to our `/etc/passwd` file. The fifth column is designated for user info, most commonly the user's first and last names. In my example, the fifth field is blank for `jdoe`, as I created `jdoe` with the `useradd` command, which didn't prompt me for the first and last names. This field is also nicknamed the `GECOS` field, and you may see it referred to as such when you read the documentation.

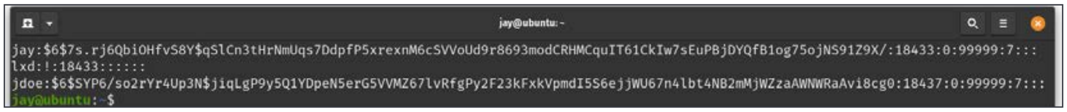
In the sixth column, the home directory for each user is shown. In the case of `jdoe`, it's set as `/home/jdoe`. Finally, we designate the user's shell as `/bin/bash`. This field refers to the default shell the user will use, which defaults to `/bin/bash` when an account is created with the `adduser` command, and `/bin/sh` when created with the `useradd` command. (If you have no preference, `/bin/bash` is the best choice for most.) If we want the user to use a different shell, we can clarify that here (though shells other than `/bin/bash` are beyond the scope of this book). If we wanted, we could change the user's shell to something invalid to prevent them from logging in at all. This is useful for when a security issue requires us to disable an account quickly.

## Using /etc/shadow

With that out of the way, let's take a look at the `/etc/shadow` file. We can use `cat` to display the contents like any other text file, but unlike `/etc/passwd`, we need `root` privileges in order to view it. So, go ahead and display the contents of this file, and I'll walk you through it:

```
sudo cat /etc/shadow
```

This will display the following output:



```
jay:$6$7s.rj6QbiOHfvS8Y$qSlCn3tHrNmUqs7DdpfP5xrexnM6cSVVoUd9r8693modCRHMCquIT61CkIw7sEuPBjDYQfB1og75ojNS91Z9X/:18433:0:99999:7:::
lxd:!:18433::::::
jdoe:$6$SYP6/so2rYr4Up3N$jiqLgP9y5Q1YDpeN5erG5VVMZ67lvRfgPy2F23kFxkVpmdI5S6ejjWU67n4lbt4NB2mMjWZzaAWNWRaAvi8cg0:18437:0:99999:7:::
jay@ubuntu:~$
```

Figure 2.6: Example /etc/shadow file

The preceding screenshot, *Figure 2.6*, shows the last three lines of this file on my server. First, we have the username in the first column—no surprises there. Note that the output is not showing the UID for each user in this file. The system knows which username matches to which UID based on the `/etc/passwd` file, so there's no need to repeat that here. In the second column, we have what appears to be random gobbledygook. Actually, that's the most important part of this entire file. That's the actual hash for the user's password.

> Although the concept is beyond the scope of the book, a password hash is a conversion of the actual password to a different string that represents the original password. This is a one-way conversion, so you cannot find the actual password by reverse-engineering the hash. In the `/etc/passwd` file, the hash of the password is stored rather than the actual password, for security purposes.

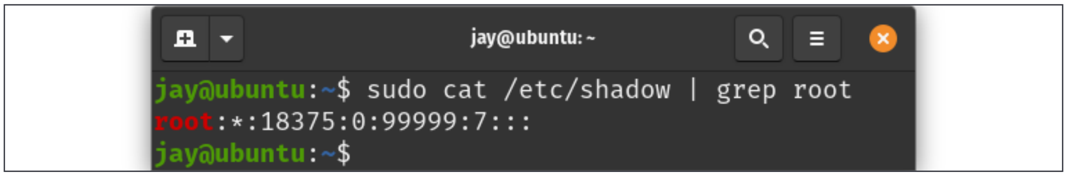
If you recall, in the `/etc/passwd` file, each user listing had an `x` for the second column, and I mentioned I would explain that later. What the `x` refers to is the fact that the user's password is encrypted and simply not stored in `/etc/passwd`. It is instead stored in `/etc/shadow`. After all, the `/etc/passwd` file is viewable by everyone, so it would compromise security quite a bit if anyone could just open up the file and see what everyone's password hashes were.

In the days of old, you could actually store a user's password in `/etc/passwd`, but it's never done that way anymore. Whenever you create a user account on a modern Linux system, the user's password is encrypted (an `x` is placed in the second column of `/etc/passwd` for the user), and the actual password hash is stored in the second column of `/etc/shadow` to keep it away from prying eyes. Hopefully, now the relationship between these two files has become apparent.

Remember earlier I mentioned that the `root` user account is locked out by default? Well, let's actually see that in action. Execute the following command to see the `root` user account entry in `/etc/shadow`:

```
sudo cat /etc/shadow | grep root
```

On my system, I get the following output:



```
jay@ubuntu:~$ sudo cat /etc/shadow | grep root
root:*:18375:0:99999:7:::
jay@ubuntu:~$
```

Figure 2.7: Example /etc/shadow file

You should notice right away that the `root` user account doesn't have a password hash at all. Instead, there's an asterisk where the password hash would've been. In practice, placing an asterisk or exclamation point here is one way to lock an account. Even easier, you can use the `passwd -l` command to lock an account without having to edit a file. But either way, we can still switch to the `root` account (which I'll show you how to do later on in this chapter). Entering an asterisk or exclamation mark in the second field creates the restriction that we can't directly log in as that user from the shell or over the network. We have to log in to the system as a normal user account first, and then we can still switch to that user if we want to.

With the discussion of password hashes out of the way, there are a few more fields within `/etc/shadow` entries that we should probably understand. Here's a contrived example line.

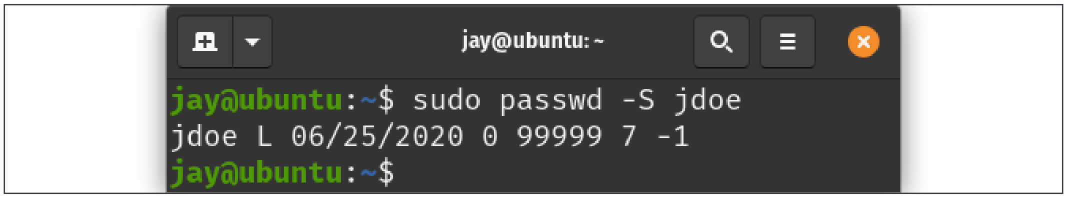
```
mulder:$6$TPxx8Z.:16809:0:99999:7:::
```

Continuing on with the third column, we can see the number of days since the Unix epoch that the password was last changed. For those that don't know, the Unix epoch is January 1, 1970. Therefore, we can read that column as the password having last been changed 16,809 days after the Unix epoch.

Personally, I like to use the following command to show more easily when the password was last changed:

```
sudo passwd -S <username>
```

This will result in an output that looks something like the following:



```
jay@ubuntu:~$ sudo passwd -S jdoe
jdoe L 06/25/2020 0 99999 7 -1
jay@ubuntu:~$
```

Figure 2.8: Checking the date of the last password change for a user

By executing this command, you can view information about any account on your system. The first column is obviously the username. The second has to do with the status of the password, which in this case is `L`, which refers to the fact that the user has a password that is locked. It would show `P` if the password was set and usable, or `NP` if the user didn't have a password at all.

The third column of this command's output gives you the actual date of the last password change for the user. The fourth column tells us how many days are required to pass before the user will be able to change their password again. In this example, `jdoe` can change the password any time because the minimum number of days is set to `0`. We'll talk about how to set the minimum number of days later on in this chapter, but I'll give you a brief explanation of what this refers to. At first, it may seem silly to require a user to wait a certain number of days to be able to change their password. However, never underestimate the laziness of your users. It's quite common for a user, when required to change their password, to change their password to satisfy history requirements, only to then just change it back to what it was originally. By setting a minimum number of days, you're forcing a waiting period in between password changes, making it less convenient for your users to cycle back through to their original password.

The fifth column, as you can probably guess, is the maximum number of days that can pass between password changes. If you require your users to change their passwords every certain number of days, you'll see that in this column. By default, this is set to `99999` days. That number of days is way beyond the human lifespan, so it may as well be infinite.

Continuing with the sixth column, we have the number of days that will elapse before the expiration date on which the user is warned that they will soon be required to change their password. In the seventh column, we set how many days can pass after the password expires, in which case the account will be disabled. With our example user, this is not set. Finally, with the eighth column (which is not visible), we can see the number of days since the Unix epoch that will elapse before the account is disabled (in our case, there's nothing here, so there is no disabled day set).

We'll go over setting these fields later, but for now, hopefully you understand the contents of the `/etc/shadow` file better.

If at any time you'd like additional clarification, feel free to check out the Ubuntu man pages. A man page (short for manual page) can give you quite a bit more information about commands as well as files. For example, the following command shows you the man page for the `ls` command:

```
man ls
```

More specific to this section, you can retrieve man pages for the `/etc/shadow` files as well:

```
man passwd
man shadow
```

Press *q* on your keyboard to exit out of a man page. Feel free to check out the man pages for any command in this book to learn more as you go along.

Now that we fully understand how to manage our users, we can also look at how to provide them with default files in their home directory.

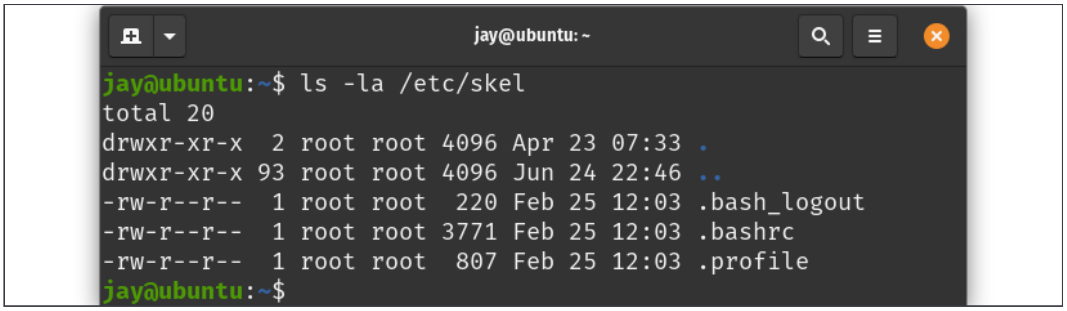
## Distributing default configuration files with /etc/skel

In a typical organization, there are usually some defaults that are recommended for users in terms of files and configuration. For example, in a company that performs software development, there are likely recommended settings for text editors and version control systems. Files that are contained within `/etc/skel` are copied into the home directory for all new users when you create them (assuming you've chosen to create a home directory while setting up the user).

In fact, you can see this for yourself right now. Execute the following command:

```
ls -la /etc/skel
```

Now, you should be able to view the contents of the `/etc/skel` directory:



```
jay@ubuntu:~$ ls -la /etc/skel
total 20
drwxr-xr-x  2 root root 4096 Apr 23 07:33 .
drwxr-xr-x 93 root root 4096 Jun 24 22:46 ..
-rw-r--r--  1 root root  220 Feb 25 12:03 .bash_logout
-rw-r--r--  1 root root 3771 Feb 25 12:03 .bashrc
-rw-r--r--  1 root root  807 Feb 25 12:03 .profile
jay@ubuntu:~$
```

Figure 2.9: Default /etc/skel files

You probably already know how to list files within a directory, but I added the `-a` option because I wanted to view hidden files as well. The files included in `/etc/skel` by default are hidden (their filenames begin with a period). I threw in the `-l` parameter solely because it shows a long list, which I think is easier to read.

Each time you create a new user and request a home directory to be created as well, these three files, shown in *Figure 2.9*, will be copied into their home directory, along with any other files you create here. You can verify this by listing the storage of the home directories for the users you've created so far. The `.bashrc` file in one user's home directory should be the same as any other, unless they've made changes to it.

Armed with this knowledge, it should be extremely easy to create default files for new users that you create. For example, you could create a file named `welcome` with your favorite text editor and place it in `/etc/skel`. Perhaps you may create this file to contain helpful phone numbers and information for new hires in your company. The file would then be automatically copied to the new user's home directory when you create the account. The user, after logging in, would see this file in his or her home directory and see the information. More practically, if your company has specific editor settings that are favored for writing code, you can include those files in `/etc/skel` as well to help ensure your users are compliant. In fact, you can include default configuration files for any application your company uses.

Go ahead and give it a try. Feel free to create some random text files and then create a new user afterward, and you'll see that these files will propagate into the home directories of the new user accounts that you add to your system.

Now that we have multiple users and have also seen how to manage their default files, we can take a look at how to switch from one user to another.

## Switching users

Now that we have several users on our system, we need to know how to switch between them. Of course, you can always just log in to the server as one of the users, but you can actually switch to any user account at any time, provided you either know that user's password or have `sudo` access.

The command you will use to switch from one user to another is the `su` command. If you enter `su` with no options, it will assume that you want to switch to `root` and will ask you for your `root` password. As I mentioned earlier, Ubuntu locks the `root` account by default, so at this point you may not have a `root` password.

> Even though Ubuntu doesn't create a password for `root` by default, some **Virtual Private Server** (**VPS**) providers unlock the `root` password and actually have you log in as the `root` user. Having an unlocked `root` account is not a standard Ubuntu practice, and is a customization specific to some cloud providers.

Unlocking the `root` account is actually really simple; all you have to do is create a `root` password. To do that, you can execute the following command as any user with `sudo` access:

```
sudo passwd
```

The command will ask you to create and confirm your `root` password. From this point on, you will be able to use the `root` account as any other account. You can log in as `root`, switch to `root`—it's fully available now. You really don't have to unlock the `root` account in order to use it. You certainly can, but there are other ways to switch to `root` without unlocking it, and it's typically better to leave the `root` account locked unless you have a very specific reason to unlock it. The following command will allow you to switch to `root` from a user account that has `sudo` access:

```
sudo su -
```

Now you'll be logged in as `root` and will be able to execute any command you want with no restrictions whatsoever. To return to your previous logged-in account, simply type `exit`. You can tell which user you're logged in as by the value at the beginning of your `bash` prompt. What if you want to switch to an account other than `root`? Of course, you can simply log out and then log in as that user. But you really don't have to do that. The following command will do the job, providing you know the password for the account:

```
su - <username>
```

The shell will ask for that user's password and then you'll be logged in as that user. Again, type `exit` when you're done using the account, which will return you to the one you were using before you switched.

That command is all well and good if you know the user's password, but you often won't. Typically, in an enterprise, you'll create an account, force the user to change their password at first login, and then you will no longer know that user's password. Since you have `root` and `sudo` access, you could always change their password and then log in as them. But they'll know something is amiss if their password suddenly stops working—you're not eavesdropping, are you? Armed with `sudo` access, you can use `sudo` to change to any user you want to, even if you don't know their password. Just prefix our previous command with `sudo` and you'll only need to enter the password for your user account, instead of theirs:

```
sudo su - <username>
```

Switching to another user account is often very helpful for support (especially while troubleshooting permissions). As an example, say that a user comes to you complaining that he or she cannot access the contents of a specific directory, or they are unable to run a command. In that case, you can log in to the server, switch to their user account, and try to reproduce their problem. That way, you can not only see their problem yourself, but you can also test out whether or not your fix has solved their issue before you report back to them.

Now we have a full understanding of user accounts, and even how to switch between them. In the next section, we'll look into groups, which allow us to categorize our users.

## Managing groups

Now that we understand how to create, manage, and switch between user accounts, we'll need to understand how to manage **groups** as well. The concept of groups in Linux is not very different from other platforms and pretty much serves the exact same purpose. With groups, you can more efficiently control a user's access to resources on your server. By assigning a group to a resource (a file, a directory, and so on), you can allow and disallow access to users by simply adding them or removing them from the group.

The way this works in Linux is that every file or directory has both a user and a group that takes ownership of it. This is contrary to platforms such as Windows, which can have multiple groups assigned to a single resource. With Linux, it's just one-to-one ownership: just one user and just one group assigned to each file or directory. If you list the contents of a directory on a Linux system, you can see this for yourself:

```
ls -l
```

The following is a sample line of output from a directory on one of my servers:

```
-rw-r--r-- 1 root bind  490 2020-04-15 22:05 named.conf
```

In this case, we can see that `root` owns the file and that the group `bind` is also assigned to it. Ignore the other fields for now; I'll explain them later when we get to the section of this chapter dedicated to permissions. For now, just keep in mind that one user and one group are assigned to each file or directory.

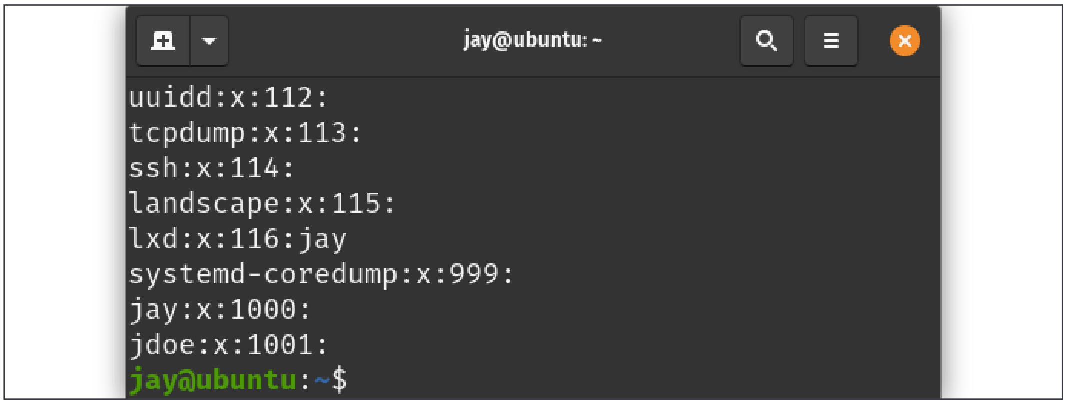
While each file or directory can only have one group assignment, any user account can be a member of any number of groups. Entering the `groups` command by itself with no options will tell you what groups your logged-in user is currently a member of. If you add a username to the `groups` command, you'll see which groups that user is a member of. Go ahead and give the `groups` command a try with and without providing a username to get the idea.

On the Ubuntu Server platform, you'll likely see that each of your user accounts is a member of a group that's named the same as your username. As I mentioned earlier, when you create a user account, you're also creating a group with the same name as the user. On some Linux distributions, though, a user's primary group will default to a group called `users` instead. If you were to execute the `groups` command as a user on the Ubuntu desktop platform, you would likely see additional groups. This is due to the fact that distributions of Linux that cater to being a server platform are often more stripped down and users on desktop platforms need access to more things such as printers, audio cards, and so on. Some packages that can be installed also add additional system users to the server.

If you were curious as to which groups exist on your server, all you would need to do is `cat` the contents of the `/etc/group` file. Similar to the `/etc/passwd` file we covered earlier, the `/etc/group` file contains information regarding the groups that have been created on your system. Go ahead and take a look at this file on your system:

```
cat /etc/group
```

The following is sample output from this file on one of my servers:



```
uuidd:x:112:
tcpdump:x:113:
ssh:x:114:
landscape:x:115:
lxd:x:116:jay
systemd-coredump:x:999:
jay:x:1000:
jdoe:x:1001:
jay@ubuntu:~$
```

Figure 2.10: Sample output from the /etc/group file

Like before, the columns in this file are separated by colons, though each line is only four columns long. In the first column, we have the name of the group. No surprise there. In the second, we are able to store a password for the group, but this is not used often as it's actually a security risk to do so. In the third column, we have the GID, which is similar in concept to the UID from when we were discussing users. Finally, in the last column, we (would) see a comma-separated list of each user that is a member of each of the groups. In this case, we're seeing that one user, `jay`, is a member of the `lxd` group.

Several entries don't show any group memberships at all. Each user is indeed a member of their own group, so this is implied even though it doesn't explicitly call that out in this file. If you take a look at the `/etc/passwd` entries for your users, you will see that their primary group (shown as the third column in the form of a GID) references a group contained in the `/etc/group` file.

Creating new groups on your system is easy to do and is a great idea for categorizing your users and what they are able to do. Perhaps you can create an `accounting` group for your accountants, an `admins` group for those in your IT department, and a `sales` group for your salespeople. The `groupadd` command allows you to create new groups. If you wanted to, you could just edit the `/etc/group` file and add a new line with your group information manually, although, in my opinion, using `groupadd` saves you some work and ensures that group entries are created properly. Editing group and user files directly is typically frowned upon (and a typo can cause serious problems). Anyway, what follows is an example of creating a new group with the `groupadd` command:

```
sudo groupadd admins
```

If you take a look at the `/etc/group` file again after adding a new group, you'll see that a new line was created in the file and a `GID` was chosen for you (the first one that hadn't been used yet). Removing a group is just as easy. Just issue the `groupdel` command followed by the name of the group you wish to remove:

```
sudo groupdel admins
```

Next, we'll take a look at the `usermod` command, which will allow you to actually associate users with groups. The `usermod` command is more or less a Swiss Army knife; there are several things you can do with that command (adding a user to a group is just one of its abilities). If we wanted to add a user to our `admins` group, we would issue the following command:

```
sudo usermod -aG admins myuser
```

In that example, we're supplying the `-a` option, which means append, and immediately following that we're using `-G,` which means we would like to modify secondary group membership. I put the two options together with a single dash (`-aG`), but you could also issue them separately (`-a -G`) as well. The example I gave only adds the user to additional groups, it doesn't replace their primary group.

Be careful not to miss the `-a` option here, as you will instead replace all current group memberships with the new one, which is usually not what you want. The `-a` option means append, or to add the existing list of group memberships for that user.

If you wanted to change a user's primary group, you would use the `-g` option instead (lowercase *g* instead of an uppercase *G* as we used earlier):

```
sudo usermod -g <group-name> <username>
```

Feel free to check out the man pages for the `usermod` command, to see all the nifty things it allows you to do with your users. You can peruse the man page for the `usermod` command with the following command:

```
man usermod
```

One additional example is changing a user's `/home` directory. Suppose that one of your users has undergone a name change, so you'd like to change their username, as well as moving their previous `home` directory (and their files) to a new one. The following commands will take care of that:

```
sudo usermod -d /home/jsmith jdoe -m
sudo usermod -l jsmith jdoe
```

In that example, we're moving the home directory for `jdoe` to `/home/jdoe`, and then in the second example, we're changing the username from `jdoe` to `jsmith`.

If you wish to remove a user from a group, you can use the `gpasswd` command to do so. `gpasswd -d` will do the trick:

```
sudo gpasswd -d <username> <grouptoremove>
```

In fact, `gpasswd` can also be used in place of `usermod` to add a user to a group:

```
sudo gpasswd -a <username> <group>
```

So, now you know how to manage groups. With the efficient management of groups, you'll be able to manage the resources on your server better. Of course, groups are relatively useless without some explanation of how to manage permissions (otherwise, nothing would actually be enforcing a member of a group to be allowed access to a resource). Later on in this chapter, we'll cover permissions so that you have a complete understanding of how to manage user access.

## Managing passwords and password policies

In this chapter, we've already covered a bit of password management, since I've given you a few examples of the `passwd` command. If you recall, the `passwd` command allows us to change the password of the currently logged-in user. In addition, using `passwd` as `root` (and supplying a username) allows us to change the password for any user account on our system. But that's not all this command can do.

### Locking and unlocking user accounts

One thing I've neglected to mention regarding the `passwd` command is the fact that you can use it to lock and unlock a user's account. There are many reasons why you may want to do this. For instance, if a user is going on vacation or extended leave, perhaps you'd want to lock their account so that it cannot be used while they are away. After all, the fewer active accounts, the smaller your attack surface. To lock an account, use the `-l` option:

```
sudo passwd -l <username>
```

Ad to unlock it, use the `-u` option:

```
sudo passwd -u <username>
```

However, locking an account will not prevent the user from logging in if they have access to the server via SSH while utilizing public key authentication. In that case, you'd want to remove their ability to use SSH as well. One common way of doing this is to limit SSH access to users who are members of a specific group. When you lock an account, simply remove them from the group. Don't worry so much about the SSH portion of this discussion if this is new to you. We will discuss securing your SSH server in *Chapter 21*, *Securing Your Server*. For now, just keep in mind that you can use `passwd` to lock or unlock accounts, and if you utilize SSH, you'll want to lock your users out of that to prevent them from logging in.

However, there's more to password management than the `passwd` command, as we can also implement our own policies as well, such as viewing or adjusting password expiration details.

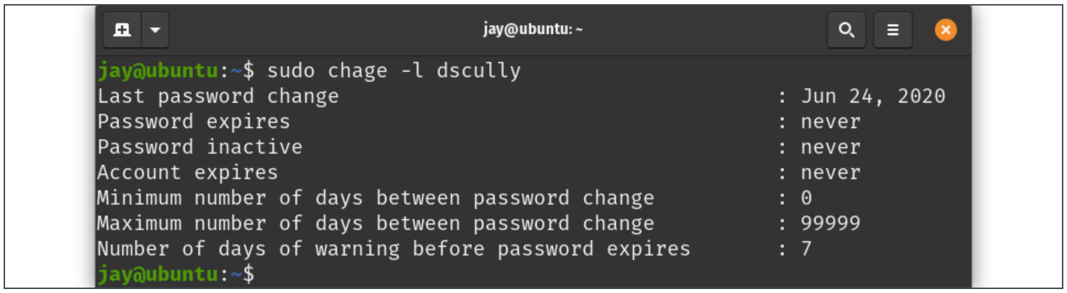
## Setting password expiration information

Earlier, I mentioned that you can set an expiration date on a user's password (during our discussion on the `/etc/shadow` file). In this section, we'll go through how to actually do that. Specifically, the `chage` command gives us this ability. We can use `chage` to alter the expiration period of a user's password, but it's also a more convenient way of viewing current expiration information than viewing the `/etc/shadow` file. With the `-l` option of `chage`, along with providing a username, we can see the relevant info:

```
sudo chage -l <username>
```

Using `sudo` or `root` is not required to run `chage`. You're able to view expiration information for your own username without needing to escalate permissions. However, if you want to view information via `chage` for any user account other than your own, you will need to use `sudo`.

In the example that follows, we can see the output of this command from a sample user account:



```
jay@ubuntu:~$ sudo chage -l dscully
Last password change                                    : Jun 24, 2020
Password expires                                        : never
Password inactive                                       : never
Account expires                                         : never
Minimum number of days between password change          : 0
Maximum number of days between password change          : 99999
Number of days of warning before password expires       : 7
jay@ubuntu:~$
```

Figure 2.11: Output from the chage command

In the output, we can see values for the date of expiration, the maximum number of days between password changes, and so on. Basically, it's the same information stored in `/etc/shadow` but it's much easier to read.

If you would like to change this information, `chage` will again be the tool of choice. The first example I'll provide is a very common one. When creating user accounts, you'll certainly want them to change their password when they first log in.

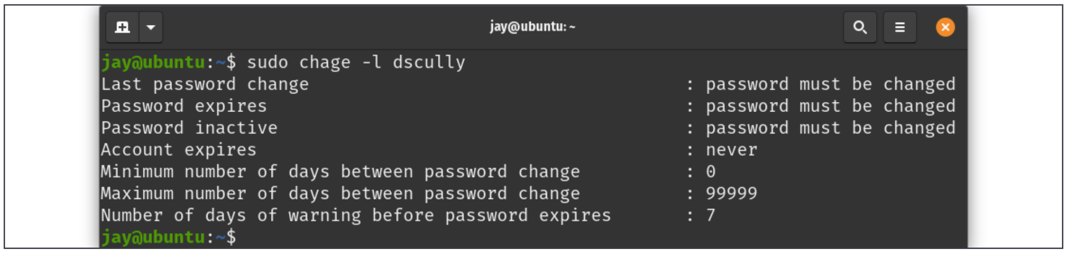
Unfortunately, not everyone will be keen on doing so. The `chage` command allows you to force a password change for a user when he or she first logs in. Basically, you can set their number of days to expiry to `0` as follows:

```
sudo chage -d 0 <username>
```

You can see the results of this command immediately if you run `chage -l` again against the user account you just modified:

```
sudo chage -l <username>
```

The output will display information regarding the password change:



```
jay@ubuntu:~$ sudo chage -l dscully
Last password change                                    : password must be changed
Password expires                                        : password must be changed
Password inactive                                       : password must be changed
Account expires                                         : never
Minimum number of days between password change          : 0
Maximum number of days between password change          : 99999
Number of days of warning before password expires       : 7
jay@ubuntu:~$
```

Figure 2.12: The chage command listing a user that has a required password change period set

To set a user account to require a password change after a certain period of days, the following will do the trick:

```
sudo chage -M 90 <username>
```

In that example, I'm setting the user account to expire and require a password change in 90 days. When the impending date reaches 7 days before the password is to be changed, the user will see a warning message when they log in.

As I mentioned earlier, users will often do whatever they can to cheat password requirements and may try to change their password back to what it was originally after satisfying the initial required password change. You can set the minimum number of days with the `-m` flag, as you can see in the next example where we set it to 5 days:

```
sudo chage -m 5 dscully
```

The trick with setting a minimum password age is to set it so that it will be inconvenient for the user to change their password to the original one, but you still want a user to be able to change their password when they feel the need to (so don't set it too long, either). If a user wants to change their password before the minimum number of days elapses (for example, if your user feels that their account may have been compromised), they can always have you change it for them. However, if you make your password requirements too much of an inconvenience, it can also work against you.

## Setting a password policy

Next, we should discuss setting a password policy. After all, forcing your users to change their passwords does little good if they change it to something simple, such as `abc123`. A password policy allows you to force requirements on your users for things such as length, complexity, and so on.

To facilitate this, we have **Pluggable Authentication Module** (**PAM**) at our disposal. PAM gives us additional functionality and control over the process of authentication, and also provides additional plugins we can use to extend authentication and add additional features. Although a full walkthrough of PAM is beyond the scope of this book, I recommend keeping it fresh in your mind in case you want to add additional features later.

Specific to the subject of setting up a password policy, we can install a PAM module to enable this, which involves installing a new package:

```
sudo apt install libpam-cracklib
```

Next, let's take a look at the following file, which is provided with Ubuntu. Feel free to open it with a text editor, such as `nano`, as we'll need to edit it:

```
sudo nano /etc/pam.d/common-password
```

An extremely important tip while modifying configuration files related to authentication (such as password requirements, `sudo` access, SSH, and so on) is to always keep a `root` shell open at all times while you make changes, and in another shell, test those changes. Do not log out of your `root` window until you are 100% certain that your changes have been thoroughly tested. While testing a policy, make sure that not only can your users log in, but your admins too. Otherwise, you may remove your ability to log in to a server and make changes.

To enable a history requirement for your passwords (meaning the system remembers the last several passwords a user has used, preventing them from reusing them), we can add the following lines to the file:

```
password   required   pam_pwhistory.so
remember=99 use_authok
```

In the example `config` line, I'm using `remember=99`, which (as you can probably guess) will cause our system to remember the last 99 passwords for each user and prevent them from using those passwords again. If you configured a minimum password age earlier, for example, 5 days, it would take the user 495 days to cycle back to their original password if you take into account that the user changes their password once every 5 days, 99 times. That pretty much makes it impossible for the user to utilize their old passwords.

Another field worth mentioning within the `/etc/pam.d/common-password` file is the section that reads `difok=3`. This configuration details that at least three characters have to be different before the password is considered acceptable. Otherwise, the password would be deemed too similar to the old one and refused. You can change this value to whatever you like; the default is normally `5` but Ubuntu defaults it to `3` in their implementation of this config file. In addition, you'll also see `obscure` mentioned in the file as well, which prevents simple passwords from being used (such as common dictionary words and so on).

Setting a password policy is a great practice to increase the security of your server. However, it's also important to not get carried away. In order to strike a balance between security and user frustration, the challenge is always to create enough restrictions to increase your security, while trying to minimize the frustration of your users. Of course, the mere mention of the word "password" to a typical user is enough to frustrate them, so you can't please everyone. But in terms of overall system security, I'm sure your users will appreciate the fact that they can be reasonably sure that you as an administrator have taken the necessary precautions to keep their (and your company's) data safe. When it comes down to it, use your best judgment.

Since we're on the subject of security, we should also take a look at configuring `sudo` itself, which we'll take care of in the next section.

## Configuring administrator access with sudo

By now, we've already used `sudo` quite a few times in this book. At this point, you should already be aware of the fact that `sudo` allows you to execute commands as if you were logged in as another user, with `root` being the default. However, we haven't had any formal discussion about it yet, nor have we discussed how to actually modify which of your user accounts are able to utilize `sudo`.

On all Linux systems, you should protect your `root` account with a strong password and limit it to be used by as few people as possible. On Ubuntu, the `root` account is locked anyway, so unless you unlocked it by setting a password, it cannot be used to log in to the system. Using `sudo` is an alternative to logging in as `root` to execute commands directly, so you can give your administrators access to perform tasks that require `root` privileges with `sudo` without actually giving them your `root` password or unlocking the `root` account. In fact, `sudo` allows you to be a bit more granular. Using `root` directly is basically all or nothing—if someone knows the `root` password and the `root` account is enabled, that person is not limited and can do whatever they want. With `sudo`, that can also be true, but you can actually restrict some users to use only particular commands and therefore limit the scope of what they are able to do on the system. For example, you could give an admin access to install software updates but not allow them to reboot the server.

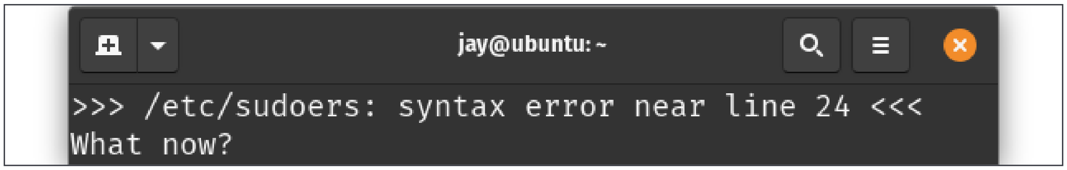
By default, members of the `sudo` group are able to use `sudo` without any restrictions. Basically, members of this group can do anything `root` can do (which is everything). During installation, the user account you created was made a member of `sudo`. To give additional users access to `sudo`, all you would need to do is add them to the `sudo` group:

```
sudo usermod -aG sudo <username>
```

> Not all distributions utilize the `sudo` group by default, or even automatically install `sudo`. Other distributions require you to install `sudo` manually and may use another group (such as `wheel`) to govern access to `sudo`.

But again, that gives those users access to everything, and that may or may not be what you want. To actually configure `sudo`, we use the `visudo` command. This command assists you with editing `/etc/sudoers`, which is the configuration file that governs `sudo` access. Although you can edit `/etc/sudoers` yourself with a text editor, configuring `sudo` in that way is strongly discouraged. The `visudo` command checks to make sure your changes follow the correct syntax and helps prevent you from accidentally destroying the file. This is a very good approach, because if you did make any errors in the `/etc/sudoers` file, you may wind up in a situation where no one is able to gain administrative control over the server. And while there are ways to recover from such a mistake, it's certainly not a very pleasant situation to find yourself in! So, the takeaway here is never to edit the `/etc/sudoers` file directly; always use `visudo` to do so.

Here's an example of the type of warning the `visudo` command shows when you make a mistake:



```
>>> /etc/sudoers: syntax error near line 24 <<<
What now?
```

Figure 2.13: The visudo command showing an error

If you do see this error, press *e* to return to edit the file, and then correct the mistake.

The way this works is when you run `visudo` from your shell, you are brought into a text editor with the `/etc/sudoers` file opened up. You can then make changes to the file and save it like you would any other text file. By default, Ubuntu opens up the `nano` text editor when you use `visudo`. With `nano`, you can save changes using *Ctrl* + *w*, and you can exit the text editor with *Ctrl* + *x*.

So `visudo` allows you to make changes to who is able to access `sudo`. But how do you actually make these changes? Go ahead and scroll through the `/etc/sudoers` file that `visudo` opens and you should see a line similar to the following:

```
%sudo   ALL=(ALL:ALL) ALL
```

This is the line of configuration that enables `sudo` access to anyone who is a member of the `sudo` group. You can change the group name to any that you'd like, for example, perhaps you'd like to create a group called `admins` instead. If you do change this, make sure that you actually create that group and add yourself and your staff to be members of it before you edit the `/etc/sudoers` file or log off; it would be rather embarrassing if you found yourself locked out of administrator access to the server.

Of course, you don't have to enable access by group. You can actually call out a username instead. With the `/etc/sudoers` file, groups are preceded by `%`, while users are not. As an example of this, we also have the following line in the file:

```
root    ALL=(ALL:ALL) ALL
```

Here, we're calling out a username (in this case, `root`), but the rest of the line is the same as the one I pointed out before. While you can certainly copy this line and paste it one or more times (substituting `root` for a different username) to grant access to others, using the group approach is really the best way. It's easier to add and remove users from a group (such as the `sudo` group) than it is to use `visudo` each time.

So, at this point, you're probably wondering what the options on `/etc/sudoers` configuration lines actually mean. So far, both examples used `ALL=(ALL:ALL) All`. In order to fully understand `sudo`, understanding the other fields is extremely important, so let's go through them (using the `root` line again as an example).

The first `ALL` means that `root` is able to use `sudo` from any terminal. The second `ALL` means that `root` can use `sudo` to impersonate any other user. The third `ALL` means that `root` can impersonate any other group. Finally, the last `ALL` refers to what commands this user is able to do; in this case, any command he or she wishes.

To help drive this home, I'll give some additional examples. Here's a hypothetical example:

```
charlie    ALL=(ALL:ALL) /sbin/reboot,/sbin/shutdown
```

Here, we're allowing user `charlie` to execute the `reboot` and `shutdown` commands. If user `charlie` tries to do something else (such as install a package), they will receive an error message:

```
Sorry, user charlie is not allowed to execute '/usr/bin/apt install
tmux' as root on ubuntu.
```

However, if `charlie` wants to use the `reboot` or `shutdown` commands on the server, they will be able to do so because we explicitly called out those commands while setting up this user's `sudo` access. We can limit this further by changing the first `ALL` to a machine name, in this case, `ubuntu`, to reference the host name of the server I'm using for my examples. I've also changed the command that `charlie` is allowed to run:

```
charlie    ubuntu=(ALL:ALL) /usr/bin/apt
```

It's always a good idea to use full paths to commands when editing `sudo` permissions, rather than the shortened versions. For example, we used `/usr/bin/apt` here, instead of just `apt`. This is important, as the user could create a script named `apt` to do mischievous things that we normally wouldn't allow them to do. By using the full path, we're limiting the user to the binary stored at that path.

Now, `charlie` is only able to use `apt`. They can use `apt update`, `apt dist-upgrade`, and any other sub-command of `apt`. But if they try to reboot the server, remove protected files, add users, or anything else we haven't explicitly set, they will be prevented from doing so.

We have another problem, though. We're allowing `charlie` to impersonate other users. This may not be completely terrible given the context of installing packages (impersonating another user would be useless unless that user also has access to install packages), but it's bad form to allow this unless we really need to. In this case, we could just remove the `(ALL:ALL)` from the line altogether to prevent `charlie` from using the `-u` option of `sudo` to run commands as other users:

```
charlie    ubuntu= /usr/bin/apt
```

On the other hand, if we actually do want `charlie` to be able to impersonate other users (but only specific users), we can call out the username and group that `charlie` is allowed to act on behalf of by setting those values:

```
charlie    ubuntu=(dscully:admins) ALL
```

In that example, `charlie` is able to run commands on behalf of the user `dscully` and the group `admins`.

Of course, there is much more to `sudo` than what I've mentioned in this section. Entire books could be written about `sudo` (and have been), but 99% of what you will need for your daily management of this tool involves how to add access to users while being specific about what each user is able to do. As a best practice, use groups when you can (for example, you could have an `apt` group, a `reboot` group, and so on) and be as specific as you can regarding who is able to do what. This way, you're not only able to keep the `root` account private (or even better, disabled), but you also have more accountability on your servers.

Now that we've explored granting access to `sudo`, we will next take a look at permissions, which give us even more control over what our users are able to access.

## Setting permissions on files and directories

In this section, all the user management we've done in this chapter so far all comes together. We've learned how to add accounts, manage accounts, and secure them, but we haven't actually done any work regarding managing the resources as far as who is able to access them. In this section, I'll give you a brief overview of how permissions work in Ubuntu Server and then I'll provide some examples for customizing them.

### Viewing permissions

I'm sure by now that you understand how to list the contents of a directory with the `ls` command. When it comes to viewing permissions, the `-l` flag is especially handy, as the output that the long listing provides allows us to view the permissions of an object:

```
ls -l
```

The following are some example, hypothetical file listings:

```
-rw-rw-rw- 1 doctor   5        Jan 11   12:52 welcome
-rw-r--r-- 1 root     root   665      Feb 19   2014 profile
-rwxr-xr-x 1 dalek   dalek   35125    Nov  7   2013 exterminate
```

In each line, we see several fields of information. The first column is our permission string for the object (for example, `-rw-r—r--`), which we'll cover in more detail shortly. We also see the link count for the object (second column). Links are beyond the scope of this chapter but will be discussed in *Chapter 5*, *Managing Files and Directories*. Continuing on, the user that owns the file is displayed in the third column, the group that owns the file is in the fourth column, the size in bytes is in the fifth, the last date the file was modified is in the sixth, and finally there is the name of the file.

Keep in mind that depending on how your shell is configured, your output may look different and the fields may be in different places. For the sake of our discussion on permissions, what we're really concerned with is the permissions string, as well as the owning user and group. In this case, we can see that the first file (named `welcome`) is owned by a user named `doctor`. The second file is named `profile` and is owned by `root`. Finally, we have a file named `exterminate` owned by a user named `dalek`.

With these files, we have the permission strings of `-rw-rw-rw-`, `-rw-r--r--`, and `-rwxr-xr-x` respectively. If you haven't worked with permissions before, these may seem strange, but it's actually quite easy when you break them down. Each permission string can be broken down into four groups, as I'll show you in the following table:

| **Object type** | **User** | **Group** | **World** |
|---|---|---|---|
| `-` | `rw-` | `rw-` | `rw-` |
| `-` | `rw-` | `r--` | `r--` |
| `-` | `rwx` | `rwx` | `r-x` |

I've broken down each of the three example permission strings into four groups. Basically, I split them each at the first character and then again every third. The first section of a permission string is just one character. In each of these examples, it's just a single hyphen. This refers to what type the object is. Is it a directory? A file? A link? In our case, each of these permission strings is a file, because the first positions of the permission strings are all hyphens. If the object were a directory, the first character would've been a `d` instead of a `-`. If the object were a link, this field would've been `l` (lowercase *L*) instead.

In the next section, in the second column of each object, we have three characters, `rw-`, `rw-`, and `rwx` respectively. This refers to the permissions that apply to the user that owns the file. For example, here is the first permission string again:

```
-rw-rw-rw- 1 doctor doctor    5 Jan 11 12:52 welcome
```

The third section of the preceding code output shows us that `doctor` is the user that owns the file. Therefore, referring back to the table, the second column of the permission string (`rw-`) applies specifically to the user `doctor`. Moving on, the third column of permissions is also three characters; in this case, `rw-` again. This section of the permissions string refers to the group that owns the file. In this case, the group is also named `doctor`, as you can see in column four of the preceding code output. Finally, the last section of the permission string, visualized in the table (`rw-` yet again, in this case), refers to world, also known as other. This basically refers to anyone else other than the owning user and owning group. Therefore, literally everyone else gets at least `rw-` permissions on the object.

Individually, `r` stands for *read* and `w` stands for *write*. Therefore, we can read the second column (`rw-`), indicating that the user (`doctor`) has access to read and write to this file. The third column (`rw-` again) tells us the `doctor` group also has read and write access to this file. The fourth column of the permission string is the same, so anyone else would also have read and write permissions to the file as well.

The third permission string I gave as an example looks a bit different. Here it is again:

```
-rwxr-xr-x 1 dalek dalek      35125 Nov  7  2013 exterminate
```

Here, we see the `x` attribute set. The `x` attribute refers to the ability to execute the file as a script. So, with that in mind, we know that this file is likely a script and is executable by users, groups, and others. Given the filename of `exterminate`, this is rather suspicious, and if it were a real file, we'd probably want to look into it.

If a permission is not set, it will simply be a single hyphen where there would normally be `r`, `w`, or `x`. This is the same as indicating that a permission is disabled. Here are some examples:

- `rwx`: Object has read, write, and execute permissions set for this field
- `r-x`: Object has read enabled, write disabled, and execute enabled for this field
- `r--`: Object has read enabled, write disabled, and execute disabled for this field
- `---`: Object has no permissions enabled for this field

Bringing this discussion all the way home, here are a few more permission strings:

```
-rw-r--r-- 1   sue   accounting      35125  Nov  7  2013 budget.txt
drwxr-x--- 1   bob   sales           35125  Nov  7  2013 annual_
projects
```

For the first of these examples, we see that `sue` is the owning user of `budget.txt` and that this file is assigned an accounting group. This object is readable and writable by `sue` and readable by everyone else (group and world). This is probably bad, considering this is a budget file and is probably confidential. We'll change it later.

The `annual_projects` object is a directory, which we can tell from the `d` in the first column. This directory is owned by the `bob` user and the `sales` group. However, since this is a directory, each of the permission bits has different meanings. In the following two tables, I'll outline the meanings of these bits for files and again for directories:

- Files:

| **Bit** | **Meaning** |
|---|---|
| `r` | The file can be read |
| `w` | The file can be written to |
| `x` | The file can be executed as a program |

- Directories:

| **Bit** | **Meaning** |
|---|---|
| `r` | The contents of the directory can be viewed |
| `w` | Contents of the directory can be altered |
| `x` | The user or group can use cd to go inside the directory |

As you can see, permissions are read differently depending on their context: whether they apply to a file or a directory. In the example of the `annual_projects` directory, `bob` has `rwx` permissions to the directory. This means that the user `bob` can do everything (view the contents, add or remove contents, and use `cd` to move the current directory of his shell into the directory). Regarding a group, members of the `sales` group are able to view the contents of this directory and `cd` into it. However, no one in the `sales` group can add or remove items to or from the directory. On this object, other has no permissions set at all. This means that no one else can do anything at all with this object, not even view its contents.

## Changing permissions

So, now we understand how to read permissions on files and directories. That's great, but how do we alter them? As I mentioned earlier, the `budget.txt` file is readable by everyone (other). This is not good because the file is confidential. To change permissions on an object, we will use the `chmod` command. This command allows us to alter the permissions of files and directories in a few different ways.

First, we can simply remove read access from the `sue` user's budget file by removing the read bit from the other field. We can do that with the following example:

```
chmod o-r budget.txt
```

If we are currently not in the directory where the file resides, we need to give a full path:

```
chmod o-r /home/sue/budget.txt
```

If you're using the `chmod` command against files other than those you own yourself, you'll need to use `sudo`.

But either way, you probably get the idea. With this example, we're removing the `r` bit from other (`o-r`). If we wanted to add this bit instead, we would simply use `+` instead of `-`. Here are some additional examples of `chmod` in action:

- `chmod u+rw <filename>`: The object gets `rw` added to the `user` column
- `chmod g+r <filename>`: The owning group is given read access
- `chmod o-rw <filename>`: Other is stripped of the `rw` bits

In addition, you can also use octal point values to manage and modify permissions. This is actually the most common method of altering permissions. I like to think of this as a scoring system. That's not what it is, but it makes it a lot easier to understand to think of each type of access as having its own value. Basically, each of the permission bits (`r`, `w`, and `x`) has its own octal equivalent, as follows:

- Read: `4`
- Write: `2`
- Execute: `1`

With this style, there are only a few possibilities for numbers you can achieve when combining these octal values (each can only be used once). Therefore, we can get `0`, `1`, `2`, `3`, `4`, `5`, `6`, and `7` by adding (or not adding) these numbers in different combinations. Some of them you'll almost never see, such as an object having write access but not read. For the most part, you'll see `0`, `4`, `5`, `6`, and `7` used with `chmod` most often. For example, if we add `Read` and `Write`, we get `6`. If we add `Read` and `Execute`, we get `5`. If we add all three, we get `7`. If we add no permissions, we get `0`. We repeat this for each column (`User`, `Group`, and `Other`) to come up with a string of three numbers. Here are some examples:

- `600`: User has read and write (4+2). No other permissions are set.

  This is the same as `-rw-------`.
- `740`: User has read, write, and execute. Group has read. Other has nothing.

  This is the same as `-rwxr-----`.
- `770`: Both user and group have full access (read, write, execute). Other has nothing.

  This is the same as `-rwxrwx---`.
- `777`: Everyone has everything.

  This is the same as `-rwxrwxrwx`.

Going back to `chmod`, we can use this numbering system in practice:

- `chmod 600 filename.txt`
- `chmod 740 filename.txt`
- `chmod 770 filename.txt`

Hopefully you get the idea. If you wanted to change the permissions of a directory, the `-R` option may be helpful to you. This makes the changes recursive, meaning that you'll not only make the changes to the directory but all files and directories underneath it in one shot:

```
chmod 770 -R mydir
```

While using -R with chmod can save you some time, it can also cause trouble if you have a mix of directories and files underneath the directory you're changing permissions on. The previous example gives permissions 770 to mydir and all of its contents. If there are files inside, they are now given executable permissions to the user and group, since 7 includes the execute bit (value of 1). This may not be what you want. We can use the find command to differentiate these. While find is out of the scope of this chapter, it should be relatively simple to see what the following commands are doing and how they may be useful:

```
find /path/to/dir/ -type f -exec chmod 644 {} \;
find /path/to/dir/ -type d -exec chmod 755 {} \;
```

Basically, in the first example, the find command is locating all files (-type f) in /path/to/dir/ and everything it finds, it executes chmod 644 against. The second example is locating all directories in this same path and changing them all to permission 755. The find command isn't covered in detail here because it easily deserves a chapter of its own, but I'm including it here because hopefully these examples are useful and will be handy for you to include in your own list of useful commands.

## Changing the ownership of objects

Finally, we'll need to know how to change the ownership of files and directories. It's often the case that a particular user needs to gain access to an object, or perhaps we need to change the owning group as well. We can change user and group ownership of a file or directory with the chown command. As an example, if we wanted to change the owner of a file to sue, we could do the following:

```
sudo chown sue myfile.txt
```

In the case of a directory, we can also use the -R flag to change the ownership of the directory itself, as well as all the files and directories it may contain:

```
sudo chown -R sue mydir
```

If we would like to change the group assignment to the object, we would follow the following syntax:

```
sudo chown sue:sales myfile.txt
```

Notice the colon separating the user and the group. With that command, we established that we would like the sue user and the sales group to own this resource. Again, we could use -R if the object were a directory and we wanted to make the changes recursive.

Another command worth knowing is the `chgrp` command, which allows you to directly change the group ownership of a file. To use it, you can execute the `chgrp` command along with the group you'd like to own the file, followed by the username. For example, our previous `chown` command can be simplified to the following, since we were only modifying the group assignment of that file:

```
# sudo chgrp sales myfile.txt
```

Just like the `chown` command, we can use the `-R` option with `chgrp` to make our changes recursively, in the case of a directory.

Well, there you have it. You should now be able to manage permissions of the files and directories on your server. If you haven't worked through permissions on a Linux system before, it may take a few tries before you get the hang of it. The best thing for you to do is to practice. Create some files and directories (as well as users) and manage their permissions. Try to remove a user's access to a resource and then try to access that resource as that user anyway and see what errors you get. Fix those errors and work through more examples. With practice, you should be able to get a handle on this very quickly.

## Summary

In Linux administration and related fields, managing users and permissions is something you'll find yourself doing quite a bit. New users will join your organization, while others will leave, so this is something that will become ingrained in your mental toolset. Even if you're the only person using your servers, you'll find yourself managing permissions for applications as well, given the fact that processes cannot function if they don't have access to their required resources.

In this chapter, we took a lengthy dive into managing users, groups, and permissions. We worked through creating and removing users, assigning permissions, and managing administrative access with `sudo`. Practice these concepts on your server. When you get the hang of it, I'll see you in our next chapter, where we'll discuss all things related to package management. It's going to be epic.

## Further reading

- User management (Ubuntu documentation): `https://ubuntu.com/server/docs/security-users`.
- File permissions (Ubuntu community wiki): `https://help.ubuntu.com/community/FilePermissions`.

# 3
# Managing Software Packages

Now that you have a server installation set up, and you know how to manage users on it, it's time to cover the management of software. The Ubuntu platform has a huge range of software available, featuring packages for everything from server administration to games. In fact, as of the time I'm writing this chapter, there are over 60,000 packages in Ubuntu's repositories. That's a lot of software packages, and in this chapter, we'll take a look at how to manage them. We'll cover how to install, remove, and update packages, as well as the use of related tools.

As we go through these concepts, we will cover:

- Understanding Linux package management
- Understanding the differences between Debian and Snap packages
- Installing and removing software
- Searching for packages
- Managing package repositories
- Backing up and restoring Debian packages
- Cleaning up orphaned APT packages
- Taking advantage of hardware enablement updates

To get started, let's build an understanding of how software packages are distributed in Ubuntu, and the basic principles of the concept of package management itself.

## Understanding Linux package management

Nowadays, *app stores* are all the rage on most platforms; typically, you'll have one central location from which to retrieve applications, allowing you to install them on your device. Even phones and tablets utilize a central software repository in which software is curated and made available. The Android platform has Google Play, Apple offers its App Store, and so on. For Linux users, this concept isn't new. The concept of software repositories is similar to that of app stores and has been around within the Linux community since long before cellular phones even had color screens.

Linux has had package management since the '90s, initially popularized by **Debian** and then **Red Hat**. Software repositories are generally made available in the form of **mirrors**, to which your server subscribes. Mirrors are available across a multitude of geographic areas, so, typically, your installation of Ubuntu Server would subscribe to the mirror closest to you. These mirrors are populated with software packages that you'll be able to install. Many packages depend on other packages, so various tools on the Linux platform exist to automatically handle these dependencies for you. Not all distributions of Linux feature package management and dependency resolution, but Ubuntu certainly does, benefiting from the groundwork already built by Debian.

Packages contained within these mirrors are constantly changing. Traditionally, an individual known as a **package maintainer** is responsible for one or more packages, and ships new versions to the repositories for approval and, eventually, distribution to mirrors. Specific to Ubuntu's repositories, a group of developers is responsible for maintaining packages rather than just a single maintainer. Most of the time, the new version of a package is provided in order to patch a security vulnerability, but otherwise it contains no new features. With the majority of Ubuntu's packages being open source, anyone is able to look at the source code, find problems, and report issues. When vulnerabilities are found, the responsible team will review the claim and then release an updated version to correct it. This process happens very quickly, and I've seen severe vulnerabilities patched even on the same day that they were reported in some cases. Ubuntu developers are definitely on top of their game in terms of taking care of security issues.

New versions of packages are also sometimes feature updates, which are updates released to introduce new features not necessarily tied to a security vulnerability. This could be a new version of a desktop application such as Firefox or a server package such as MySQL. Most of the time, though, new versions of packages that are vastly different are held for the next Ubuntu release. The reason for this is that too much change can cause your server to not be as stable, and experience application or even entire OS-level crashes. Instead, known working and stable packages are preferred; however, given the fact that Ubuntu releases every six months, you don't have to wait very long.

Specifically, feature updates in a **long-term support** (**LTS**) release such as Ubuntu 20.04 must go through an approval process before being made available in the default repositories, and these feature updates are specifically referred to as **Stable Release Updates** (**SRUs**). There's an entire process around how these updates are approved, but the general take away is that there should be a good reason to implement a major version change in a stable, long-term Ubuntu release.

As a server administrator, you'll often need to make a choice between security and feature updates. Security updates are the most important of all and allow you to patch your servers in response to security vulnerabilities. Sometimes, feature updates become required in your organization because it's decided that new features may benefit you or may become required for current objectives. In this chapter, we won't focus on installing security updates (we'll take care of that in *Chapter 21, Securing Your Server*), but it's important to understand the reasons new packages are made available to you.

Package management is typically very convenient in Ubuntu, with security updates and bug fixes coming regularly. With just one command (which we'll get to shortly), you can install a package along with all of its dependencies. Having performed manual dependency resolution myself, I can tell you first-hand that having dependencies handled automatically is a wonderful thing. The main benefit of how packages are maintained on a Linux server is that you generally don't have to search the internet for packages to download, as Ubuntu's repositories contain most of the ones you'll ever need. As we continue through this chapter, you'll come to know everything you need in order to manage this software.

## Understanding the differences between Debian and Snap packages

Now, before we actually get into the ins and outs of managing packages, there are actually two completely different types of packages available to you, and you should understand the differences between them. As of the time of writing, we're at a kind of crossroads regarding the way in which software is managed in Linux.

Traditionally, each distribution has its own package format, and its own utilities to manage them. Ubuntu utilizes **Debian packages** (with package names ending in `.deb`) as the main package format, which Ubuntu inherits from the Debian distribution (Ubuntu is forked from Debian, which means that it uses Debian as a foundation). Ubuntu and Debian utilize the `apt` and `dpkg` commands to manage packages. On the other hand, distributions such as CentOS and Red Hat use **RPM packages** for their distributions, and the `dnf` command to manage them. First, let's discuss Debian packages.

## Debian packages

**Debian packages** have been the main type of package in Ubuntu for the entire existence of the distribution so far. When you search online for how to install something in Ubuntu, chances are, you're going to be installing a Debian package. These packages are known as Debian packages because Ubuntu is built from Debian sources and utilizes the same commands in order to install these packages. So even though Ubuntu is not Debian (Debian is a completely different distribution), they both use the same package format primarily.

The naming may be confusing for newcomers, because if Ubuntu is considered a different distribution than Debian, then why refer to its packages as "Debian" packages? Debian packages have a filename that ends in `.deb`, and this package format originated in Debian. Ubuntu didn't develop its own package type; it uses the same package format as Debian. Therefore, whether we are installing packages in Debian or Ubuntu using a command such as `apt`, Debian packages are the type of packages used for both.

If you've worked with Ubuntu before reading this book, then chances are, you've already used the `apt` series of commands to carry out some package management. Debian packages are great, because when paired with the `apt` command, they are easy to use and handle dependency resolution for you. However, they present some challenges and major drawbacks.

First, the majority of the distribution is made up of Debian packages. This means that the Linux kernel, system packages, libraries, and security updates are all Debian packages that are installed when you install Ubuntu Server. When you install security updates, Debian packages are installed. The reason this may be a problem is that other software you'll be installing, such as Apache, MariaDB, and so on, are also Debian packages, and may conflict with system packages when one package requires a pre-requisite package that conflicts with another. This can lead to a situation where you can't install packages at all. Package maintainers are generally good at avoiding this scenario, so conflicts aren't incredibly common nowadays.

However, with Debian packages, if a system library gets corrupted, literally every piece of software that depends on it will fail. Ubuntu developers pay a great deal of attention to this, so you shouldn't run into issues. But the truth is, this is a lot of work for the maintainers of Ubuntu to deal with.

Another concern with Debian packages is software availability. When a new major version of a package is released, it generally will not be offered to you until the next release of the distribution. This means that if you want a version of PHP, Apache, or some other piece of software that's newer than what your current release of Ubuntu features, you generally will not have it offered to you. Instead, you typically wait until the next release of the entire distribution. There are some exceptions to this, such as Firefox in the desktop version of Ubuntu. As mentioned before, new major package versions are exceptions to the rule and come from the SRU approval process. While having tried-and-true software available that has been extensively tested offers better stability, sometimes you may require a newer version of particular software than what is available, which may lead you to consider alternative sources. After all, you don't have to install a new version of Windows or macOS just to install newer applications!

**Universal packages** are a new concept for Linux, and are intended to be a single package format that multiple distributions recognize. The idea is that a developer only has to compile one package (instead of a separate package for each distribution) and users would only need to download a single package regardless of their chosen Linux flavor. The next section will discuss a type of universal package called Snaps.

## Snap packages

As it stands today, developers would need to create multiple package types to support Linux. Perhaps they'll create Debian packages for Debian itself and Ubuntu, and then RPM packages for CentOS, Red Hat Enterprise Linux, and SuSE. And while you may think that having to create two package types isn't all that bad, consider that each of the RPM-based distributions needs its own specific RPM package, and you can see how it might be tedious to a developer to have to create five or more different packages for any one release of their software.

As a result of this, there's a push to adopt a single package format that each distribution can install that is independent of the system package type. This concept is known as **universal packages**, and the idea is to have a standard package type that can be installed on any Linux distribution. As a result, developers would only have to create one package to have their application work on all the popular distros.

The Ubuntu community commonly uses the word *distro* as a contraction of the term distribution.

Another benefit of universal packages is that they have all of their dependencies built in, so conflicts are less likely to occur; everything the application needs would be contained in one single package. This is great, because the likelihood of you running into package conflicts with universal packages is next to non-existent.

As with all things in the IT industry, we can't collectively be satisfied and decide on the proper technology. As such, there is debate among the community regarding which one of the multiple types of universal packages is the most suitable. I won't get into the political debates in this book, as each has its strengths and weaknesses. But even easier for us in regards to Ubuntu Server, only one of them is ideal for server installations anyway. The competing technologies for universal packages include Flatpak, AppImage, and Snap packages.

Canonical, the makers of Ubuntu, understand the pain points that developers and users experience, and have been making a great effort to change how packages are managed. The type of universal package they've developed to address these concerns is known as the **Snap package**. Like all universal packages, Snap packages (or more simply, *Snaps*) have no impact on the underlying Debian packages at all and are completely independent, thus removing the possibility of conflicts with your system packages. This also allows you to have a newer version of an application installed than what would otherwise be made available. Since Snaps are installed separately and independently from the underlying Debian packages, there's no reason to withhold them. Snap packages are better in just about every way and are a great concept. The only downside might be that they are larger packages, since they include not only the application itself but also all the libraries they require in one single package. However, they're really not that large and shouldn't cause an issue with disk space. These packages are no bigger than a typical application on macOS or Microsoft Windows.

So, which should you choose? It really just depends on your use case. Each of the universal package types is good at some things but has downsides. For the purpose of servers, Snaps are superior considering that Flatpaks and AppImages typically target only desktop applications. Flatpaks and AppImages are also great technologies, but where they fall behind for our use case is that they don't support server (non-**Graphical User Interface**, or non-**GUI**) applications well. This means that these package types are best for installing applications you'd normally find on a desktop Ubuntu installation, such as music players, browsers, graphical text editors, and so on. Snap packages target *both* GUI and non-GUI apps. Since server installations normally don't contain a GUI at all, this makes our choice for us. It could be the case that Flatpaks and AppImages start supporting non-GUI apps better in the future, but for now, the choice of universal package format is simple.

As it stands today, though, the majority of the packages we'll be installing are going to be Debian packages, as Snap packages are a bit slow to catch on. They're making steady progress, but it remains to be seen how well the industry will embrace them. For now, it's probably best to evaluate Snap packages when they're available and to choose the most appropriate package type based on availability, security, version, and support.

With the differences between Snaps and Debian packages out of the way, let's work through some examples of how to actually manage the software on our server. We'll look at commands to search for available packages, and then we'll install them and remove them.

## Installing and removing software

Before we begin, we will want to research a bit regarding the application we want to install. In Ubuntu, there are multiple ways of installing software, so the best way to find out how to get started is by simply checking the documentation on the website for the application we want to install. Typically, a Google search will do (just make sure you check the domain and are on the correct site). Most software will have installation instructions for various platforms, including Ubuntu. Most of the time, it will just tell you to download the Debian package format via the `apt install` command. Other times, the software may have a Snap available, or even a PPA repository (we'll discuss PPA repositories later on in this chapter). Let's start our package management journey by taking a look at the `apt` commands that are used to install Debian packages.

## Managing Debian packages with apt

**APT**, or **Advanced Package Tool**, is a suite of tools that allows us to install, remove, and update Debian packages. There are various sub-commands that make up this suite, which we'll go over now. The most popular variation of the `apt` command is `apt install`. If you've ever read instructions for how to do something in Ubuntu, chances are you've already run that command to install a package. And that's exactly what it does: it allows you to install packages for Ubuntu over the command line. For example, the following command will install the `openssh-server` package:

```
sudo apt install openssh-server
```

You can also install multiple packages at a time by separating each with a space, instead of installing each package one at a time. The following example will install three different packages:

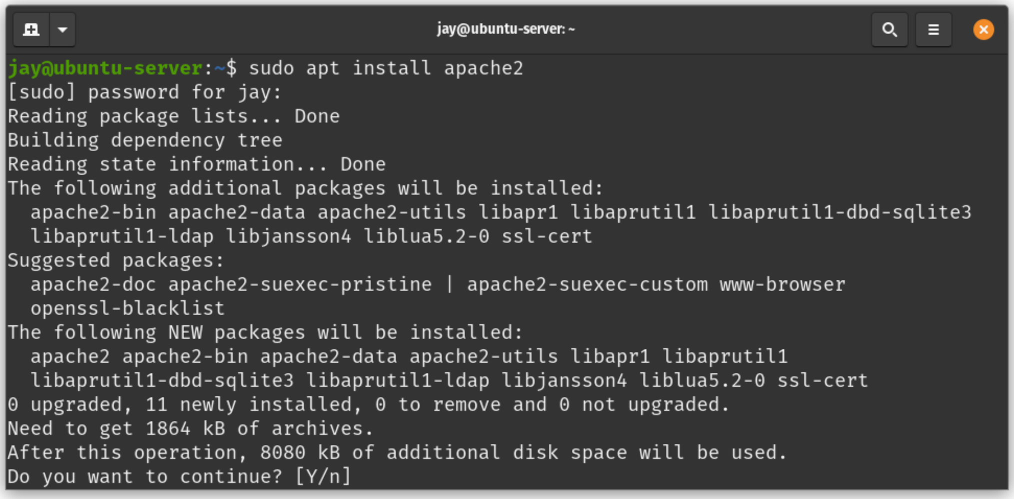
```
sudo apt install <package1> <package2> <package3>
```

In some books, blogs, and articles, you may see longer versions of `apt` commands such as `apt-get install` instead of just `apt install`. Being able to shorten commands such as `apt-get install` to `apt install` is a relatively new feature of `apt` in Debian and Ubuntu. Both methods are perfectly valid, but simplifying APT commands down to just `apt` is preferred going forward.

So, what actually happens when you install a package with `apt`? If you've run through the process before, you're probably accustomed to this process already. But, typically, the process begins with `apt` calculating dependencies. The majority of packages require other packages to function, so `apt` will check to ensure that the package you're requesting is available and that its dependencies are available as well. First, you'll see a summary of the changes that `apt` wants to make to your server. In the case of installing the `apache2` package on an unconfigured Ubuntu Server, I enter the following command:

```
sudo apt install apache2
```

I see the following output on my system when it starts to install:



```
jay@ubuntu-server:~$ sudo apt install apache2
[sudo] password for jay:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  apache2-bin apache2-data apache2-utils libapr1 libaprutil1 libaprutil1-dbd-sqlite3
  libaprutil1-ldap libjansson4 liblua5.2-0 ssl-cert
Suggested packages:
  apache2-doc apache2-suexec-pristine | apache2-suexec-custom www-browser
  openssl-blacklist
The following NEW packages will be installed:
  apache2 apache2-bin apache2-data apache2-utils libapr1 libaprutil1
  libaprutil1-dbd-sqlite3 libaprutil1-ldap libjansson4 liblua5.2-0 ssl-cert
0 upgraded, 11 newly installed, 0 to remove and 0 not upgraded.
Need to get 1864 kB of archives.
After this operation, 8080 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Figure 3.1: Installing Apache on a sample server

Even though I only asked for `apache2`, `apt` informs me that it also needs to install `apache2-bin`, `apache2-data`, `apache2-utils`, and `libapr1` (and others) in order to satisfy the dependencies for the `apache2` package. `apt` also suggests that I install `apache2-doc`, `apache2-suexec-pristine`, and a few others, though those are optional and are not required. You can install the suggested packages by adding the `--install-suggests` option to the `apt install` command, but that isn't always a good idea as it may install a large number of packages that you may not need. You can, of course, cherry-pick the suggested packages individually by using the `apt` command to install any one or more of them manually.

Most of the time, though, you probably won't want to do this; it's usually better to keep your installed packages to a lean minimum and install only the packages you need. As we'll discuss in *Chapter 21*, *Securing Your Server*, the fewer packages you install, the smaller the attack surface of your server.

Another option that is common with installing packages via `apt` is the `-y` option, which assumes *yes* to the confirmation prompt where you choose if you want to continue or not. For example, my previous example output included the line `Do you want to continue? [Y/n]`. If we used `-y`, the command would have proceeded to install the package without any confirmation. This can be useful for administrators in a hurry, though I personally don't see the need for this unless you are scripting your package installations. In fact, it can sometimes be a bad idea, because by assuming *yes*, you may be confirming something you'd regret.

Another neat default in Ubuntu Server is that it automatically configures most packages to have their daemons start up and also be enabled so that they start with each boot. Using the earlier example of `apache2`, the `apache2` service will start and the application will automatically start running as soon as you install the package. This may seem like a good idea for the sake of convenience, but not everyone prefers this automation. As I've mentioned, the more packages installed on your server, the higher the attack surface, but running services (also known as daemons, or units) are each a method of entry for miscreants should there be a security vulnerability. Therefore, some distributions don't enable and start daemons automatically when you install packages. The way I see it, though, you should only install packages you actually intend to use, so it stands to reason that if you go to the trouble of manually installing a package such as Apache, you probably want to start using it.

When you install a package with the `apt` keyword, it searches its local database for the package you named. If it doesn't find it, it will throw an error. Sometimes, this error may be because the package isn't available or perhaps the version that `apt` wants to install no longer exists. Ubuntu's repositories move very fast; new versions of packages are added almost daily. When a new version of a package is added, its older equivalent may be removed. For this reason, it's recommended that you update your package sources from time to time. Doing so is easy, using the following command:

```
sudo apt update
```

This command doesn't actually update any packages; it merely checks in with your local mirror to see if any packages have been added or removed and updates your local index. This command is useful because installations of packages can fail if your sources aren't up to date. In most cases, the symptom will be that `apt` errors out of the process when it can't find a package it's looking for.

Removing packages is also very easy and follows a very similar syntax; you would only need to replace the `install` keyword with `remove`:

```
sudo apt remove <package>
```

And, just like with the `install` option, you can remove multiple packages at the same time. The following example will remove three packages:

```
sudo apt remove <package1> <package2> <package3>
```

If you'd like to not only remove a package but also wipe out its configuration, you can use the `--purge` option:

```
sudo apt remove --purge <package>
```

This will not only remove the package, but wipe out its configuration directory (applications typically store their configuration files in a sub-directory of `/etc`).

So, that concludes the basics of managing Debian packages with `apt`. Now, let's move on to managing Snaps.

## Managing Snap packages with snap

To manage Snap packages, we use the `snap` command. The `snap` command features several options we can use to search for, install, and remove Snap packages from our server or workstation. To begin, we can use the `snap find` command along with a keyword to display a list of Snap packages available to us that match that keyword:

```
snap find <keyword>
```

Basically, you just simply type the `snap find` command along with a search term to look for. One example could be the `nmap` application, which is a useful tool to have if we're managing a network:

```
snap find nmap
```

In the case of `nmap`, this utility is available in Ubuntu's default repositories, so you don't need to use the Snap package to install it. Typically, though, the Snap version will be newer and have more features than what is available in the APT repositories. If we wish to install the Snap version, we can use the following command to do so:

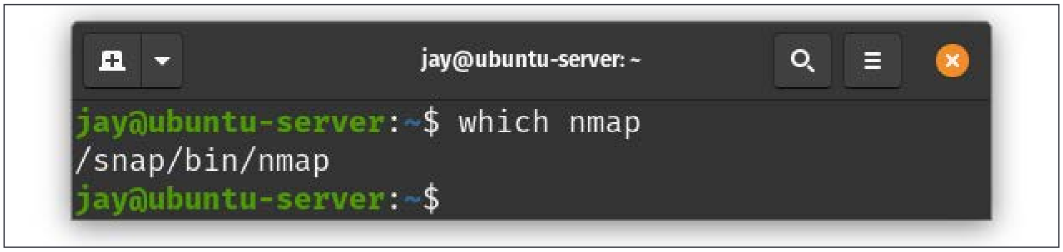
```
sudo snap install nmap
```

With the `install` option, we need to use `sudo` since the act of installing a package makes changes on the server. For a simple `find`, we can omit `sudo`.

Now that we have `nmap` installed, we can check the location of the `nmap` binary with the `which` command. We can use the `which` command to find the location of the binary file for commands that are available—it will show the path to the binary if the appropriate package is installed. If the command is not available, the `which` command will show no output. So, if we run the following command, we should now see the path of that binary printed to the screen:

```
which nmap
```

This will return the following output, showing that the Snap version of `nmap` is run from a special place:



```
jay@ubuntu-server:~$ which nmap
/snap/bin/nmap
jay@ubuntu-server:~$
```

Figure 3.2: Checking the location of the nmap binary

Now, when we run `nmap`, we're actually running it from `/snap/bin/nmap`. If we were to install `nmap` via `apt`, it would run from `/usr/bin/nmap` instead. If we also have the `nmap` utility installed from Ubuntu's APT repositories, then we can run either one at any time by calling out the full path to the binary we want to run, since Snap packages are independent of the APT packages. For example, if we had `nmap` installed via both `snap` and `apt`, we could run Ubuntu's version by running `/usr/bin/nmap` and the Snap version by running `/snap/bin/nmap`.

Removing an installed Snap package is easy. We simply use the `remove` option:

```
sudo snap remove nmap
```

If we issue that command, then `nmap` (or whichever Snap package we designate) is removed from the system.

To update a package, we use the `refresh` option along with the name of a package to update:

```
sudo snap refresh <package>
```

With that command, the package will be updated to the newest version available. Going even further, we can attempt to update every Snap on our server with the same command (without specifying a package):

```
sudo snap refresh
```

As you can see, managing Snap packages is fairly straightforward. Using the `snap` suite of commands, we can install, update, or remove packages from our server or workstation. The `snap find` command allows us to find new Snap packages to install. Perhaps as the technology matures, we may find ourselves installing more Snap packages than Debian packages, but that remains to be seen. For now, it's a nice benefit to have two options to consider when installing new software.

In addition to being able to install packages, there are some additional tips and considerations around searching for packages. After all, you can't install a package if you don't know what's available. We'll explore searching for packages in the next.

## Searching for packages

Unfortunately, the naming conventions used for packages in Ubuntu Server aren't always obvious. Worse, package names are often very different from one distribution to another even for the same piece of software. While this book and other tutorials online will outline the exact steps needed to install software, if you're ever on your own, it really doesn't help much if you don't know the name of the package you want to install. In this section, I'll try to take some of the mystery out of searching for packages.

In the previous section, we went over searching for Snap packages, so I won't repeat that here. The APT suite of utilities also has a means of searching for packages as well, which is the `apt search` command. We can use the following command to search for packages, by providing a keyword:

```
apt search <search term>
```

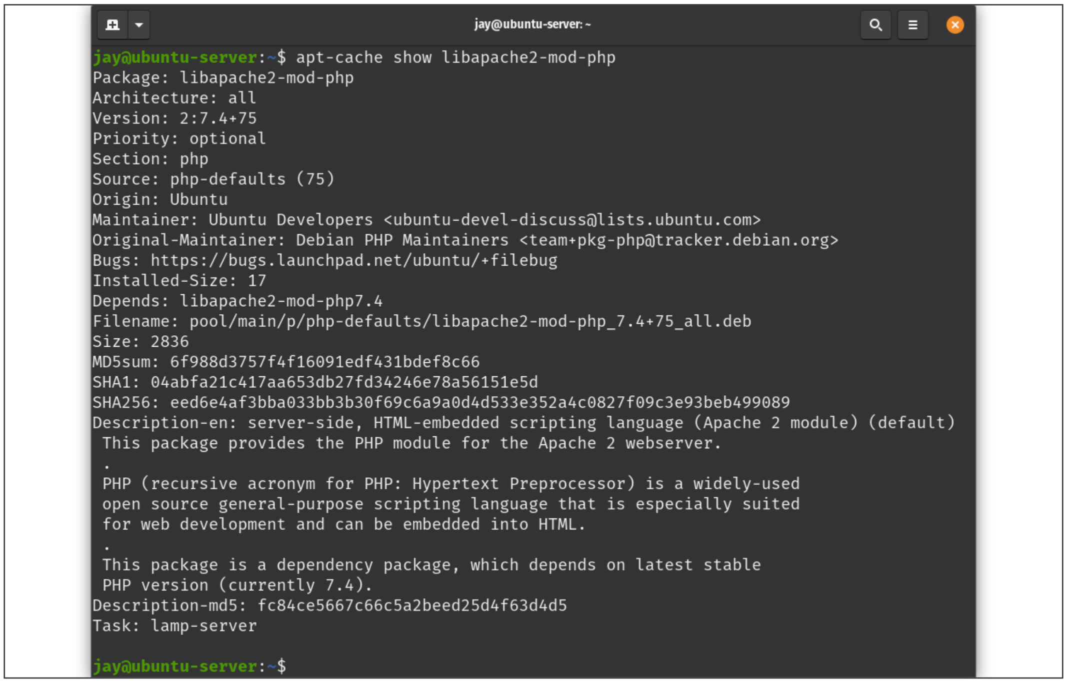
The output from this command will show a list of packages that match your search criteria, with their names and descriptions. If, for example, you wanted to install the PHP plugin for Apache and you didn't already know the name of the associated package, the following would narrow it down:

```
apt search apache php
```

In the output, we will get a list of more than a handful of packages, but we can deduce from the package descriptions in the output that `libapache2-mod-php` is most likely the one we want. We can proceed to install it using `apt`, as we would normally do. If we're not sure whether or not this is truly the package we want, we can view more information regarding this (or any other) package with the `apt-cache show` command:

```
apt-cache show libapache2-mod-php
```

The output of this command is illustrated in the following screenshot:



```
jay@ubuntu-server:~$ apt-cache show libapache2-mod-php
Package: libapache2-mod-php
Architecture: all
Version: 2:7.4+75
Priority: optional
Section: php
Source: php-defaults (75)
Origin: Ubuntu
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer: Debian PHP Maintainers <team+pkg-php@tracker.debian.org>
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Installed-Size: 17
Depends: libapache2-mod-php7.4
Filename: pool/main/p/php-defaults/libapache2-mod-php_7.4+75_all.deb
Size: 2836
MD5sum: 6f988d3757f4f16091edf431bdef8c66
SHA1: 04abfa21c417aa653db27fd34246e78a56151e5d
SHA256: eed6e4af3bba033bb3b30f69c6a9a0d4d533e352a4c0827f09c3e93beb499089
Description-en: server-side, HTML-embedded scripting language (Apache 2 module) (default)
 This package provides the PHP module for the Apache 2 webserver.
 .
 PHP (recursive acronym for PHP: Hypertext Preprocessor) is a widely-used
 open source general-purpose scripting language that is especially suited
 for web development and can be embedded into HTML.
 .
 This package is a dependency package, which depends on latest stable
 PHP version (currently 7.4).
Description-md5: fc84ce5667c66c5a2beed25d4f63d4d5
Task: lamp-server

jay@ubuntu-server:~$
```

Figure 3.3: Showing the info of a Debian package

With this command, we can see additional details regarding the package we're considering installing. In this case, we learn that the `libapache2-mod-php` package also depends on PHP itself, so that means if we install this package, we'll get the PHP plugin as well as PHP.

Another method of searching for a package (if you have a web browser available) is to connect to the *Ubuntu Packages Search* page at `http://packages.ubuntu.com/`, where you can navigate through the packages from their database for any currently supported version of Ubuntu. You won't always have access to a web browser while working on your servers, but, when you do, this is a very useful way to search through packages, view their dependencies, descriptions, and more.

Using the `apt search` command, as well as the `snap find` command, should get you quite far in the process of determining the name of the packages you want to install. Package management skills come over time, so don't expect to automatically know which packages to install right away. When in doubt, just perform a Google search, research the documentation of the software you want to run, and learn how to install it in Ubuntu. Typically, the instructions will lead you to the correct commands to use. The examples we'll go over during the course of this book will guide you through the most common use cases for Ubuntu Server.

At this point, we should have a solid understanding of the different types of packages available and how to manage them. However, it's sometimes the case that we need to run software on our server for which there is no package available within the standard repositories. Therefore, in the next section, we'll learn how to add additional repositories from which to install software.

## Managing package repositories

Often, the repositories that come pre-installed with Ubuntu will suffice for the majority of the Debian packages you'll install via APT. Every now and then, though, you may need to install an additional repository in order to take advantage of software not normally provided by Ubuntu, or versions of packages newer than what you would normally have available. Adding additional repositories allows you to subscribe to additional sources of software and install packages from them the same as you would from any other source.

Adding additional repositories should be considered a last resort, however. When you install an additional repository, you're effectively trusting the author of that repository with your organization's server. Although I haven't ever seen this happen first-hand, it's theoretically possible for authors of software to include back doors or malware in software packages (intentionally or unintentionally), and then make them available for others via a software repository. Therefore, you should only add repositories from sources that you have reason to trust.

In addition, it sometimes happens that a maintainer of a repository simply gives up on it and disappears. This I have seen happen first-hand. In this situation, the repository may go offline (which would show errors during `apt` transactions, indicating that it's not able to connect to the repository), or worse, the repository stays online, but security updates are never made available, causing your server to become vulnerable to attack. Sometimes, you just don't have a way around it. You need a specific application and Ubuntu doesn't offer it by default. Your only option may be to compile an application from source or add a repository. The decision is yours, but just keep security in mind whenever possible. When in doubt, avoid adding a repository unless it's the only way to obtain what you're looking for.

## Adding additional repositories

Software repositories are essentially URLs in a text file, stored in one of two places. The main Ubuntu repository list is stored in `/etc/apt/sources.list`. Inside that file, you'll find a multitude of repositories for Ubuntu's package manager to pull packages from. In addition, files with an extension of `.list` are read from the `/etc/apt/sources.list.d/` directory and are also used whenever you use `apt`. I'll demonstrate both methods.

A typical repository line in either of these two files will look similar to the following:

```
deb http://us.archive.ubuntu.com/ubuntu/ focal main restricted
```

The first section of each line will be either `deb` or `deb-src`, which references whether the `apt` command will find binary packages (`deb`) or source packages (`deb-src`) there. Next, we have the actual URL that `apt` will use in order to reach the repository. In the third section, we have the codename of the release; in this case, it's `focal` (which refers to the codename for Ubuntu 20.04, `Focal Fossa`).

> If you weren't already aware, the codename for each Ubuntu release is based on an animal. The fossa is an animal from Madagascar that somewhat resembles a cat but with curved ears. If you didn't already know that, now you do.

Continuing, the fourth section of each repository line refers to the `Component`, which references whether or not the repository contains software that is free and open source, and is supported officially by Canonical (the company that oversees Ubuntu's development). The component can be `main`, `restricted`, `universe`, or `multiverse`. Repositories with a `main` component include officially supported software. This generally means that the software packages have source code available, so Ubuntu developers are able to fix bugs. Software marked `restricted` is still supported but may have a questionable license. `universe` packages are supported by the community, not Canonical themselves. Finally, `multiverse` packages contain software that is neither free nor supported, which you would be using at your own risk.

As you can see from looking at the `/etc/apt/sources.list` file on your server, it's possible for a repository line to feature software from more than one component. Each repository URL may include packages from several components, and the way you differentiate them is to only subscribe to the components you need for that repository. In our previous example, the repository line included both `main` and `restricted` components. This means that, for that particular example, the `apt` utility will index both free (`main`) and non-free (`restricted`) packages from that repository.

You can add new repositories to the `/etc/apt/sources.list` file (and it will function just fine), but that's not typically the preferred method. Instead, as I mentioned earlier, `apt` will scan the `/etc/apt/sources.list.d/` directory for text files ending with the `.list` extension. These text files are formatted the same as the `/etc/apt/sources.list` file in the sense that you include one additional repository per line, but this method allows you to add a new repository by simply creating a file for it, and you can remove the repository by simply deleting that file.

This is safer than editing the `/etc/apt/sources.list` file directly, since there's always a chance you can make a typo and disrupt your ability to download packages from even the official repositories.

In addition, you may need to install a **GNU Privacy Guard** (**GnuPG**) key for a new repository, but this process differs from one application to another. Typically, the documentation will outline the entire process. This key basically protects you in that it makes sure that you're installing signed packages. Not all developers protect their applications this way, but it's definitely a good thing to do.

Once you have the repository (and possibly the key) installed on your server, you'll need to run the following command to update your package index:

```
sudo apt update
```

As mentioned earlier in this chapter, this command updates your local cache as to which packages are available on the remote server. APT is only aware of packages that are in its database, so you will need to sync this with that command before you'll be able to actually install the software contained within the repository.

## Adding Personal Package Archives

On the Ubuntu platform, there also exists another type of repository, known as a **Personal Package Archive** (**PPA**). PPAs are essentially another form of APT repository, and you'll even interact with their packages with the `apt` command, as you would normally. PPAs are usually very small repositories, often including a single application that serves a single purpose. Think of PPAs as *mini-repositories*. A PPA is common in situations where a vendor doesn't make their software available with their own repository and may only make their application available in the form of source code you would need to manually download, compile, and install. With the PPA platform, anyone can compile a package from source and easily make it available for others to download.

PPAs suffer from the same security concerns as regular repositories (you need to trust the vendor and so on), but are a bit worse considering that the software isn't audited at all. In addition, if the PPA was to ever go down, you'd stop getting security updates for the application you install from it. Only use PPAs when you absolutely need to.

There is one use case for PPAs that may be compelling, specifically, for a server platform that standard repositories aren't able to handle as well, and that is software versioning. As I mentioned earlier, a major server component such as PHP or MySQL may be locked to a specific major version with each Ubuntu Server release. What do you do if you need to use Ubuntu Server, but the application you need to run is not available in the version your organization requires? In the past, you would literally need to choose between the distribution and the package, with some organizations even using a different distribution of Linux just to satisfy the need to have a specific application at a specific version. You can always compile the application from source (assuming its source code is available), but that can cause additional headaches in the sense that you'd be responsible for compiling new security patches yourself whenever they're made available. PPAs potentially give you access to applications not normally available in the default repositories, and/or access to newer versions of packages than what is normally provided. This gives you, the server administrator, the ability to choose the approach that is best for your goal.

PPAs are generally added to your server with the `apt-add-repository` command. The syntax generally uses the `apt-add-repository` command, with a colon, followed by a username, and then the PPA name. The following command is a hypothetical example:

```
sudo apt-add-repository ppa:username/myawesomesoftware-1.0
```

To begin the process, you would start your search by visiting Ubuntu's PPA website, which is available at `https://launchpad.net/ubuntu/+ppas`. There, you can search among the PPAs available.

> Before adding a PPA to your server, it's best to first research whether or not it's being maintained well. For example, if the PPA hasn't had any updated packages in a very long time, that's cause for concern—security fixes are fairly common with most packages. If a package isn't being regularly updated and has therefore gone "stale," it may be best to avoid it as it may cause more harm than good.

Once you find a PPA you would like to add to your server, you can add it simply by finding the name of the PPA and then adding it to your server with the `apt-add-repository` command. You should take a look at the page for the PPA, though, in case there are different instructions. For the most part, the `apt-add-repository` command should work fine for you. Each PPA typically has a set of instructions attached, so there shouldn't be any guesswork required here.

So, what exactly does the `apt-add-repository` command do? Honestly, it's not all that amazing. When you install a PPA, it's essentially automating the process of adding a repository file to your `/etc/apt/sources.list.d` directory and installing its key. Therefore, you can uninstall a PPA by simply deleting its file.

PPAs are one of the things that sets Ubuntu apart from Debian and can be a very useful feature if harnessed with care. PPAs offer Ubuntu a flexible way of adding additional software that wouldn't normally be made available, though you will need to keep an eye on such repositories to ensure they are properly patched when vulnerabilities arise and are used only when absolutely necessary. Always prefer packages from Ubuntu's default repositories as well as Snaps, but PPAs offer you another option in case you can't find what you need anywhere else.

After you've maintained a server for a while, or finished setting it up for a particular goal, you'll have installed a plethora of packages to suit its purpose. Exporting a list of installed packages can make it easier to rebuild a server should the need arise, and we'll look at one method of doing that in the next section.

## Backing up and restoring Debian packages

As you maintain your server, your list of installed packages will grow. If, for some reason, you needed to rebuild your server, you would need to reproduce exactly what you had installed before, which can be a pain. It's always recommended that you document all changes made to your server via a change control process, but, at the very least, keeping track of which packages are installed is an absolute must. In some cases, a server may only include one or two extra packages in order to meet its goal, but, in other cases, you may need an exact combination of software and libraries in order to get things working like they were. Thankfully, the `dpkg` command allows us to export and import a list of packages to install.

To export a list of installed packages, we can use the following command:

```
dpkg --get-selections > packages.list
```

This command will dump a list of package selections to a standard text file. If you open it, you'll see a list of your installed packages, one per line. A typical line within the exported file will look similar to the following:

```
tmux install
```

With this list, we can import our selections back into the server if we need to reinstall Ubuntu Server, or into a new server that will serve a similar purpose. First, before we manage any packages, we should update our index:

```
sudo apt update
```

Next, we'll need to ensure we have the `dselect` package installed. The `dselect` package provides us with additional features when managing Debian packages. Its finer points are beyond the scope of this chapter, but specific to our current goal, we can use it to restore packages from our exported list. At your shell prompt, type `which dselect` and you should see output similar to the following:

```
/usr/bin/dselect
```

If you don't see the output, you'll need to install the `dselect` package with `apt`:

```
sudo apt install dselect
```

Once that's complete, you can now import your previously saved package list, and have the missing packages reinstalled on your server. The following commands will complete the process:

```
sudo dselect update
sudo dpkg --set-selections < packages.list
sudo apt-get dselect-upgrade
```

Normally, we simply use `apt` instead of `apt-get` nowadays, but oddly enough, the `dselect-upgrade` command only works with `apt-get`.

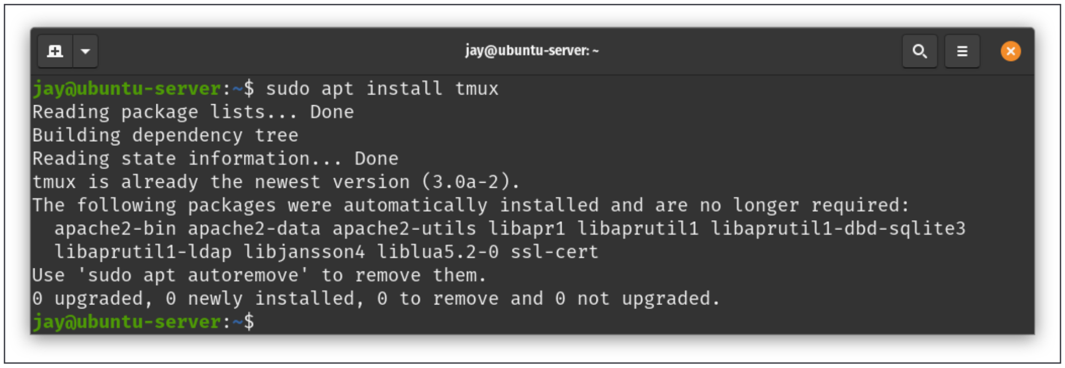
After you have run those commands, the packages that are contained in your packages list but aren't already installed will be installed once you confirm the changes. This method allows you to easily restore the packages previously installed on your server, if for some reason you need to rebuild it, as well as setting up a new server to be configured in a similar way to an existing one.

Now that we have an understanding of how to export and import a list of installed packages, we can also take a look at how to clean up unneeded packages, to ensure our server is as free as possible from unnecessary bloat.

## Cleaning up orphaned apt packages

As you manage packages on your server, you'll eventually run into a situation where you'll have packages on your system that are installed but not needed by anything. This occurs either when removing a package that has dependencies, or when the dependencies on an installed package change. As you'll remember, when you install a package that requires other packages, those dependencies are also installed. But if you remove the package that required them, the dependencies will not be removed automatically.

To illustrate this situation, if I remove the `apache2` package from one of my servers, I will see the following extra information if I then try to install something else:



```
jay@ubuntu-server:~$ sudo apt install tmux
Reading package lists... Done
Building dependency tree
Reading state information... Done
tmux is already the newest version (3.0a-2).
The following packages were automatically installed and are no longer required:
  apache2-bin apache2-data apache2-utils libapr1 libaprutil1 libaprutil1-dbd-sqlite3
  libaprutil1-ldap libjansson4 liblua5.2-0 ssl-cert
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
jay@ubuntu-server:~$
```

Figure 3.4: Output with orphaned packages shown

In this example, I removed `apache2` (that was done before the screenshot was taken), then I went on to install `tmux`. The package I was trying to install is arbitrary; the important part is the text you see in the screenshot where it says `The following packages were automatically installed and are no longer required`. Basically, if you have orphaned packages on your system, Ubuntu will remind you periodically as you use the APT suite of tools. In this case, I removed `apache2` so all of the dependencies that were installed to support the `apache2` package were no longer needed.

In the screenshot, I'm shown a list of packages that APT doesn't think I need anymore. It may be right, but this is something we would need to investigate. As instructed in the output, we can use the `apt autoremove` command as `root` or with `sudo` to remove them. This is a great way of keeping our installed packages clean but should be used with care. If you've just recently removed a package, it's probably safe to do the cleanup manually.

Although we haven't walked through updating packages (we'll do that in *Chapter 21, Securing Your Server*), a situation that may come up later is one in which you have outdated kernels that can be cleaned with the `autoremove` option. These will appear in the same way as the example orphans I was shown in the previous screenshot, but the names will contain `linux-image`. Take care with these; you should never remove outdated kernels from your server until you verify that the newly installed kernel is working correctly and the server doesn't exhibit any unwanted or unexpected behavior. Generally, you would probably want to wait at least a week before running `apt autoremove` when kernel packages are involved.

When it comes to other packages, they are generally safe to remove with the `apt autoremove` command, since the majority of them will be packages that were installed as a dependency of another package that is no longer present on the system. However, double-check that you really do want to remove each of the packages before you do so. You can always reinstall a package if you didn't mean to install it, and as an added benefit, if you reinstall a package that was marked for auto-removal, it won't show up in the output as an orphan package in the future.

As you've seen so far, there are a lot of options when it comes to managing software in Ubuntu, and as an administrator, you are able to choose the best possible method depending on the goal. Another special type of update is available, that can improve hardware support in a situation where your hardware is newer than your Ubuntu version, or provide compatibility for hardware that didn't previously have support available.

## Taking advantage of hardware enablement updates

One issue in the Linux industry has been hardware support. This is problematic in various Linux distributions because you may find yourself in a situation where you're using a server (or even a desktop or laptop) that was released with the latest processor and chipset, but no newer version of your Linux distribution has been released yet that includes updated drivers that support it. Unlike platforms such as Windows, hardware drivers are typically built right into the Linux kernel. So, if you have an old release (which would contain an older kernel), you might be out of luck for hardware support until the next version of your Linux distribution is released.

Thankfully, Ubuntu has come up with a system to address this problem, and it's one of the many things that set it apart from other distributions. Ubuntu features a set of updates known as the **hardware enablement** (**HWE**) stack, which is an exclusive feature to **long-term support** (**LTS**) releases. We discussed the difference between LTS and regular releases back in *Chapter 1*, *Deploying Ubuntu Server*. HWE updates are optional, but they add additional compatibility with newer hardware that was released after the current LTS version was made available. A new HWE stack typically includes a new kernel and driver packages. However, since drivers are often built right into the Linux kernel, you'll also get added support for things other than new video cards as well, such as newly released network cards.

HWE updates are generally made available from the second point release of an LTS version, and then again with each subsequent point release until the next LTS release becomes available. The new compatibility stack is **backported** from the most recent non-LTS release, which means that you get the same compatibility of the latest non-LTS release, while being able to stay on an LTS release. For example, when Ubuntu 18.04.2 was released on February 14, 2020, it contained the kernel and driver packages backported from Ubuntu 18.10. This means that you were able to take advantage of the newer hardware support of Ubuntu 18.10, while being able to remain on 18.04 LTS and benefit from the longer support cycle of the LTS release.

At the time of writing, Ubuntu 20.04.1 is the latest LTS release of Ubuntu, so hardware enablement hasn't been added yet. Historically, it's most likely the case that hardware enablement will play out in 20.04 LTS the same way as it has in previous releases. If that's the case, then we will have the updated packages made available once Ubuntu 20.04.2 is released.

Once the new HWE stack is made available, you can choose to install it or to remain on the original 20.04 kernel with no change. On the desktop version of Ubuntu, people that enjoy computer games can really benefit from these updates, as the new drivers enable better performance when it comes to video cards and support newer gaming hardware. When it comes to the scope of this book, the new hardware enablement benefits us a bit less than desktop users. The reason being, if your server is running fine with no issues, then there really isn't any reason to install the new hardware enablement unless the updated kernel contains new features you'd like to take advantage of.

A frustrating experience that sometimes comes up is having Ubuntu complain that there are no network cards available, even though you do actually have one. I've seen this most often when purchasing a new physical server that contains the latest hardware. It can be very frustrating to power on a brand new server, only to realize you can't get a network connection at all. This is a classic symptom of needing an updated compatibility stack. Perhaps your network card was released after the release date of the current Ubuntu release. This scenario is a good example of why hardware enablement updates exist, and why you will probably end up using them sooner or later.

If you don't opt in to the HWE updates, your server will always have the same hardware enablement (kernel, drivers, and so on) as it did when your installed LTS release was first published. In that case, your kernel and related packages will only be updated when you install new security updates. At a later date, you can opt in to HWE updates manually if you wish. Generally, you only do this if you've added new hardware to a physical server that requires a new kernel. If your server is working perfectly fine, and you haven't added new hardware, there's probably no reason to install a new HWE stack.

If you do decide to utilize these updates, there are two ways to do so. You can opt in to the newer HWE stack while installing Ubuntu Server, or you can manually install the required packages. At the time of writing, Ubuntu 20.04 is new to the scene, so it remains to be seen if the updated packages will be made available with 20.04.2, but it's very likely given past history. The following screenshot shows the first screen of the installer for Ubuntu 18.04.4. Assuming Ubuntu 20.04 follows the same plan, you'll most likely see an option such as the one in the following screenshot once 20.04.2 is released:

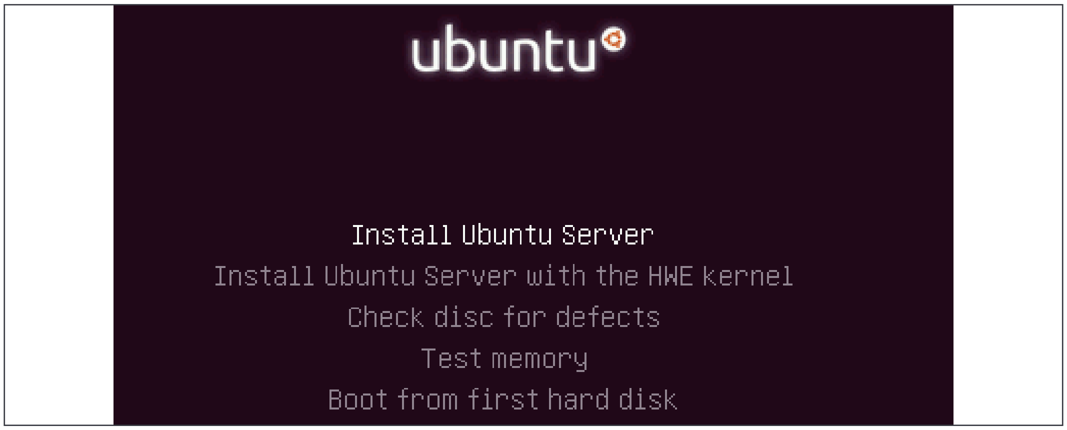

ubuntu
Install Ubuntu Server
Install Ubuntu Server with the HWE kernel
Check disc for defects
Test memory
Boot from first hard disk

Figure 3.5: Main menu of the Ubuntu installer

Notice the **Install Ubuntu Server with the HWE kernel** option. If you choose this option, then your installation of Ubuntu Server will contain the new HWE packages immediately.

If you've already installed Ubuntu Server and you'd like to install the HWE updates afterward, you can do so from the terminal. In Ubuntu 18.04, for example, you were able to switch to the HWE kernel with the following command:

```
sudo apt install --install-recommends linux-generic-hwe-18.04
```

When Ubuntu 20.04 releases its newer HWE stack, a similar command will likely be used to install it. If you need a newer HWE kernel, refer to instructions in Ubuntu's documentation pages when that time comes.

## Summary

In this chapter, we have taken a crash course in the world of package management. As you can see, Ubuntu Server offers an amazing number of software packages and various tools that we can use to manage them. We began the chapter with a discussion of how package management with Ubuntu works, then we worked through installing packages, searching for packages, and managing repositories. We have also discussed best practices for keeping our server up to date, as well as the commands available for us to install the latest updates. Snap packages were also covered, which is an exciting up-and-coming technology that will greatly enhance software distribution on Ubuntu.

In *Chapter 4*, *Navigating and Essential Commands*, we'll take a look at foundational commands for navigating the Linux Shell, understanding the filesystem layout, and more.

## Further reading

- LTSEnablementStack (Ubuntu wiki): `https://wiki.ubuntu.com/Kernel/LTSEnablementStack`
- Ubuntu Server Guide: `https://ubuntu.com/server/docs`

# 4 Navigating and Essential Commands

At this point in our journey, we've already covered a lot of ground—we've learned how to deploy an Ubuntu server, how to manage users, and most recently, how to manage software packages. Before we go too far, we should take a moment to learn some important concepts and commands that will allow us to build more of the foundational knowledge that will serve us well for the remainder of the book and beyond. These foundational concepts include core Linux commands for navigating the shell, the Linux filesystem layout, viewing the contents of files, and even checking on log files. Specifically, this discussion will include:

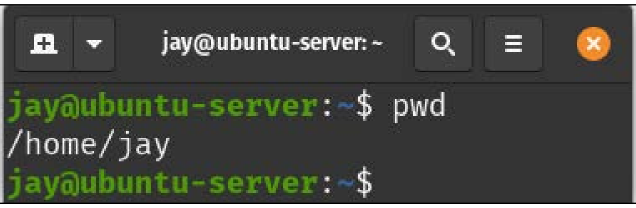
- Learning essential Linux commands
- Understanding the Linux filesystem layout
- Viewing the contents of files
- Viewing application log files

Let's take some time to learn some essential Linux commands that will help strengthen our command-line skills.

## Learning essential Linux commands

Building a solid competency on the command line is essential and effectively gives any system administrator or engineer superpowers. Our new abilities won't allow us to leap tall buildings in a single bound but will definitely enable us to execute terminal commands as if we're ninjas. While we won't master the art of using the command line in this section (that can only come with years and experience), we will definitely become more confident.

First, let's talk about moving from one place to another within the Linux filesystem. Specifically, by "Linux filesystem," I'm referring to the default structure of the various folders (also referred to as "directories") contained within your Ubuntu installation. The Linux filesystem contains many important directories, each with their own designated purpose, which we'll talk about in more detail later in this chapter. Before we can explore that further, we'll need to learn how to navigate from one directory to another. The first command we'll cover in this section relative to navigating the filesystem will clarify the directory you're currently working from. For that, we have the `pwd` command. This stands for **print working directory**, and shows you where you currently are in the filesystem. If you run it, you may see output such as this:



```
jay@ubuntu-server:~$ pwd
/home/jay
jay@ubuntu-server:~$
```

Figure 4.1: Viewing the current working directory

In this example, when I ran `pwd`, the output informed me that my current working directory is `/home/jay`. This is known as your home directory and, by default, every user has one (as we discussed in *Chapter 2*, *Managing Users and Permissions*). This is where all the files for your user account will reside by default. Sure, you can create files anywhere you'd like, even outside your home directory if you have permission to do so or you use `sudo`. But just because you *can* doesn't mean you *should*. As you'll learn in this chapter, the Linux filesystem has a designated place for just about everything. But your home directory, located at `/home/<username>`, is yours. You own it, you control it—it's your home on the server. In the early 2000s, Linux installations with a graphical user interface even depicted your home directory with an icon of a house.

Typically, files that you create in your home directory will have permission string similar to this:

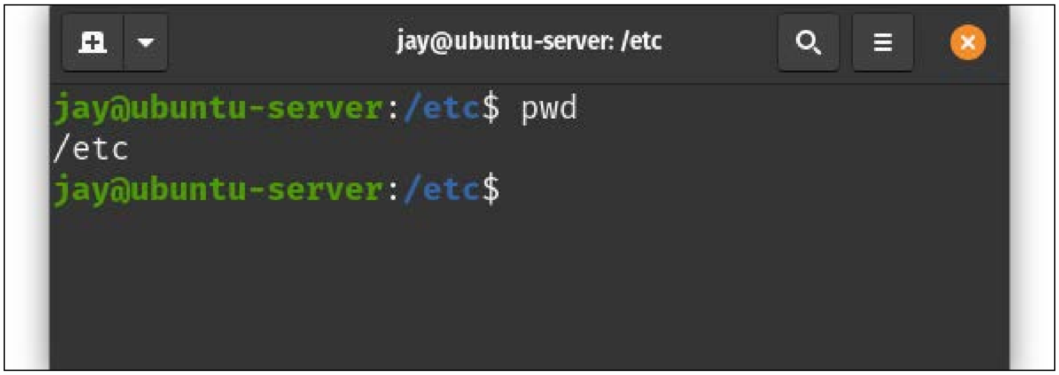
```
-rw-rw-r-- 1 jay  jay      0 Jul  5 14:10 testfile.txt
```

We've discussed permissions and gone over how to read a permission string in *Chapter 2, Managing Users and Permissions*, but you can see that by default, files you create in your home directory are owned by your user, your group, and are readable by all three categories (user, group, and other).

To change our current directory and navigate to another, we can use the `cd` command along with a path we'd like to move to:

```
cd /etc
```

Now, I haven't gone over the file and directory layout yet, so I just randomly picked the `etc` directory. The forward slash at the beginning designates the beginning of the filesystem. More on that later. Now, we're in the `/etc` directory, and our command prompt has even changed as well:



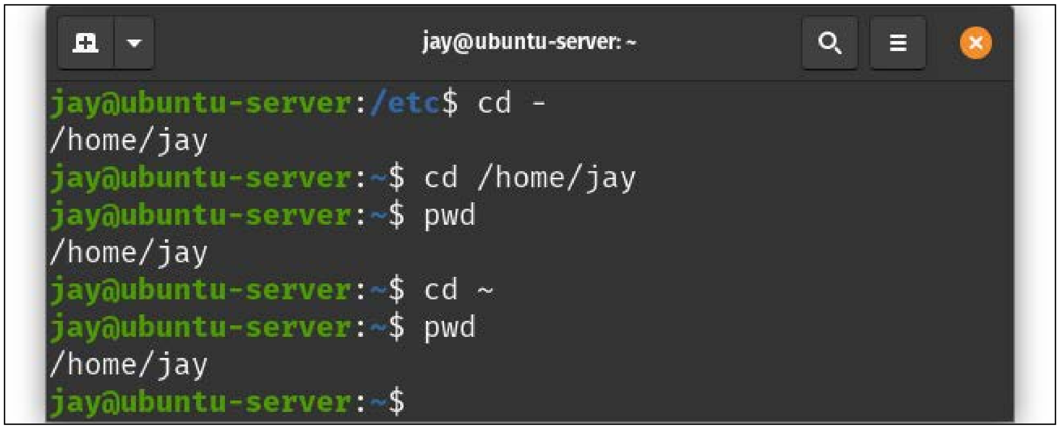
```
jay@ubuntu-server:/etc$ pwd
/etc
jay@ubuntu-server:/etc$
```

Figure 4.2: Command prompt and pwd command after changing a directory

As you could probably guess, the `cd` command stands for *change directory*, and it's how you move your working directory from one to another while navigating around. You can use the following command, for example, to return back to the home directory:

```
cd /home/<user>
```

In fact, there are several ways to return home, a few of which are demonstrated in the following screenshot:



```
jay@ubuntu-server:/etc$ cd -
/home/jay
jay@ubuntu-server:~$ cd /home/jay
jay@ubuntu-server:~$ pwd
/home/jay
jay@ubuntu-server:~$ cd ~
jay@ubuntu-server:~$ pwd
/home/jay
jay@ubuntu-server:~$
```

Figure 4.3: Other ways of navigating to the home directory

The first command, `cd -`, doesn't actually have anything to do with your home directory specifically. It's a neat trick to return you to whatever directory you were in most recently. For me, the `cd -` command took me to the previous directory I was in, which just so happened to be `/home/jay`. The second command, `cd /home/jay`, took me directly to my home directory since I called out the entire path. The last command, `cd ~`, also took me to my home directory. This is because ~ is shorthand for the full path to your home directory, so you don't really ever have to type out the entire path to `/home/<user>`. You can just refer to that path simply as ~.

Another essential command is `ls`. The `ls` command lists the contents of the current working directory. We probably don't have any contents in our home directory yet. But if we navigate to `/etc` by running `cd /etc`, as we did earlier, and then execute `ls`, we'll see that the `/etc` directory has a number of files in it. Go ahead and try it yourself and see:

```
cd /etc
ls
```

We didn't actually have to change our working directory to `/etc` just to list the contents. We could've just executed the following command:

```
ls /etc
```

Even better, we can run:

```
ls -l /etc
```

This gives us the contents in a long list, which I think is much easier to understand. It will show each directory or file entry on its own line, along with the permission string. But, you probably already remember `ls` as well as `ls -l` from back in *Chapter 2*, *Managing Users and Permissions*, so I won't go into too much more detail here. The `-l` portion of the `ls` command in that example is known as an **argument**. I'm not referring to an argument such as the ever-ensuing debate in the Linux community over which command-line text editor is the best between Vim and Emacs (it's clearly Vim). Instead, I'm referring to the concept of an argument in shell commands that allow you to override the defaults, or feed options to the command in some way, such as in this example, where we format the output of `ls` to be in a long list.

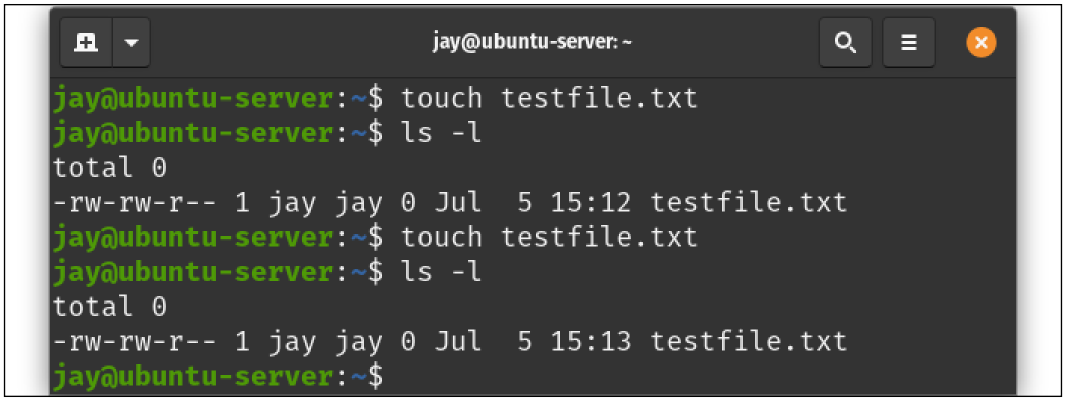
The `rm` command is another one that we touched on in *Chapter 2*, *Managing Users and Permissions*, when we were discussing manually removing the home directory of a user that was removed from the system. So, at this point, you're probably well aware of that command and what it does (it removes files and directories). It's a potentially dangerous command, as you could use it to accidentally remove something that you shouldn't have. In that chapter, we used the following command to remove the home directory of user `dscully`:

```
rm -r /home/dscully
```

As you can see, we're using the `-r` argument to alter the behavior of the `rm` command, which, by default, doesn't remove directories but only files. The `-r` argument instructs `rm` to remove everything recursively, even if it's a directory. The `-r` argument will also remove subdirectories of the path as well, so you'll definitely want to be careful with this command. As I've mentioned earlier in the book, if you use `sudo` with `rm`, you can hypothetically delete your entire Ubuntu installation!

Another option offered by `rm` is the `-f` argument which is short for *force*, and it tells `rm` not to prompt before removing things. This argument won't be needed as often, and use cases for it are outside the scope of this chapter. But keep in mind that it exists, should you need it.

Another foundational command that's good to know is `touch`, which actually serves two purposes. First, assuming you have permission to do so in your current working directory, the `touch` command will create an empty file if it doesn't already exist. Second, the `touch` command will update the modification time of a file or directory if it does already exist:



```
jay@ubuntu-server:~$ touch testfile.txt
jay@ubuntu-server:~$ ls -l
total 0
-rw-rw-r-- 1 jay jay 0 Jul  5 15:12 testfile.txt
jay@ubuntu-server:~$ touch testfile.txt
jay@ubuntu-server:~$ ls -l
total 0
-rw-rw-r-- 1 jay jay 0 Jul  5 15:13 testfile.txt
jay@ubuntu-server:~$
```

Figure 4.4: Experimenting with the touch command

To illustrate this, in the related screenshot, I ran several commands. First, I ran the following command to create an empty file:

```
touch testfile.txt
```

That file didn't exist before, so when I ran `ls -l` afterward, it showed the newly created file with a size of 0 bytes. Next, I ran the `touch testfile.txt` command again a minute later, and you can see in the screenshot that the modification time went from `15:12` to `15:13`.

When it comes to viewing the contents of a file, we'll get to that later on in this chapter. And there are definitely more commands that we'll need to learn to build the basis of our foundation. But for now, let's take a break from the foundational concepts to understand the Linux filesystem layout better, which is pretty much mandatory for some of the commands we'll learn later.

## Understanding the Linux filesystem layout

As I mentioned earlier, every directory in a Linux installation has a designated purpose. It isn't a hard rule that you have to follow, more of a strong recommendation for where certain things are supposed to go.

You can certainly go against the recommendations; ultimately, you have full control over your installation. But, if you make it a habit to place files in strange locations, you may annoy your colleagues. In this section, we'll go over the most common directories and talk about their purpose.

The term *filesystem* itself can be somewhat confusing in the Linux world because it can refer to two different things—the default directory structure, as well as the actual filesystem we choose when formatting a volume such as a hard drive or flash drive (ext4, XFS, and so on). Specific to this section, we're going to take a quick look at the Linux filesystem in the context of the default directory structure.

In Linux (Ubuntu uses the Linux kernel and related utilities, so therefore it is a *distribution* of Linux), the filesystem begins with a single forward slash, `/`. This is considered the beginning of the filesystem, and directories and sub-directories branch out from there. For example, consider the `/home` directory. This directory exists at the root level of the filesystem, which you can see from the fact that it begins with a forward slash. My home directory on my system is `/home/jay`, which means that it's the `jay` directory, which is inside the `home` directory, and that directory is at the beginning of the filesystem.

This is confusing at first, but becomes very logical once you become accustomed to it. If you're familiar with Microsoft Windows, then you can technically think of `/` as the C: drive. It's actually a bit more complex than that, but if we forego some of the quirks, that comparison works. To really bring this home, use the `ls` command against several directories on your server. If you execute `ls /`, you will see all of the directories at the root of the filesystem. You'll see the `home` directory among the results, among many others. For those of you who prefer a more visual representation, the following screenshot shows an example filesystem:

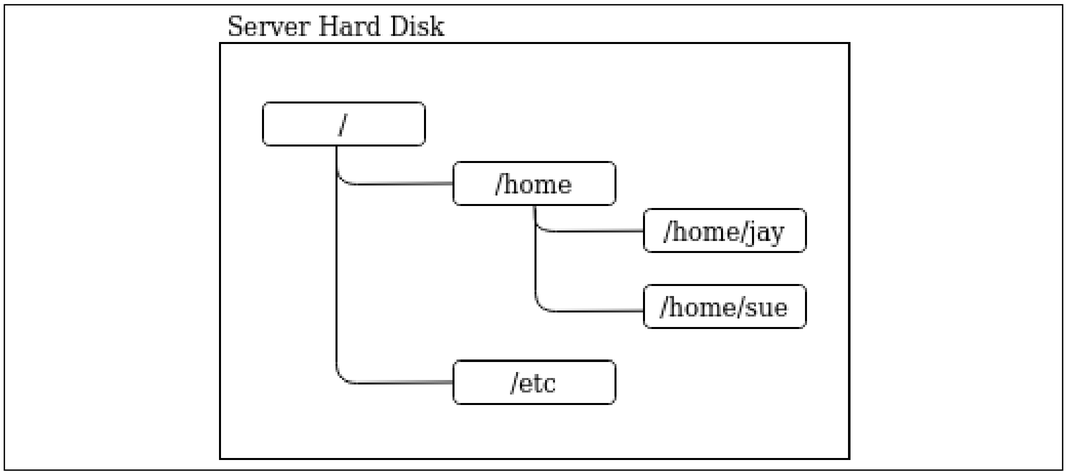


Figure 4.5: Diagram of a portion of a typical Linux filesystem

As you can see, the Linux filesystem resembles a tree, with a main branch that extends outward and directories branch out from one another. This default directory structure is part of the **Filesystem Hierarchy Standard** (**FHS**), which is a set of guidelines that defines how the directory structure is laid out. This specification defines the names of the directories, where they are located, and what they are for. Distributions will sometimes go against some of the definitions here, but for the most part, follow it fairly closely. This is why you may see a very similar (if not the same) directory structure on distributions of Linux other than Ubuntu.

So, why is this important? As I mentioned, each directory generally has a purpose. There are some debates about the default layout from time to time, and some changes are made every now and then. But in regard to Linux, the filesystem layout tends to change less frequently than other things do.

A full walkthrough of the FHS is beyond the scope of this book, but I have included a link to this specification at the end of the chapter should you decide to read more about it. There are some directories you definitely should know, however. Here are some of the more important ones.

| **Directory** | **Purpose** |
|---|---|
| `/` | The beginning of the filesystem; all directories are underneath this |
| `/etc` | System-wide application configuration |
| `/home` | User home directories |
| `/root` | The home directory for root (root doesn't have a directory under `/home`) |
| `/media` | For removable media, such as flash drives |
| `/mnt` | For volumes that are intended to stay mounted for a while |
| `/opt` | Additional software packages<br>(some programs are installed here, not as common) |
| `/bin` | Essential user binaries (`ls`, `cp`, and so on) |
| `/proc` | Virtual filesystem for OS-level components |
| `/usr/bin` | A majority of user commands |
| `/usr/lib` | Libraries |
| `/var/log` | Log files |

The `/etc` directory deserves some additional discussion, as you'll no doubt be working with it a lot. This directory, as mentioned in the preceding table, holds configuration files for applications that are intended to be respected system-wide. For example, if you're running the OpenSSH daemon on your server, you're listening for connections via port 22 by default. (Don't worry, we'll discuss these individual concepts in further chapters.) The configuration file for the OpenSSH server is located in the `/etc/ssh` directory. Since OpenSSH is a service that runs on the system as a background process, its configuration is stored in the `sshd_config` file inside that directory. If you remove the package responsible for providing OpenSSH, the configuration is retained in that file (removing packages doesn't remove config files by default), so if you go to reinstall OpenSSH later, you'll have the same configuration the next time. If we want to wipe out the configuration while uninstalling a package, we can use the `--purge` option with the `apt remove` command to do so, as we saw in *Chapter 3*, *Managing Software Packages*.

Other directories of importance will be discussed as we progress through the topics in this book. Don't worry if some of this doesn't make sense right now; it will come with time. The main point here is that there are many directories, each having their own purpose. If you want to know the purpose of a particular directory, consult the FHS. If you're curious where you should place something on a server, also consult the FHS. But again, don't worry too much about doing research outside of this book, as we'll cover the necessary topics as we go along.

We now have a better understanding of the default filesystem layout, and the purpose for some common directories. In the next section, we'll explore how we can view the contents of files stored in those directories.

## Viewing the contents of files

The Linux filesystem contains many directories and files. In the case of files, we need to learn how to read and manipulate them to round out our knowledge. We'll cover more topics surrounding file management in the next chapter. For now, we can benefit by taking a look at how to view the content of existing files.

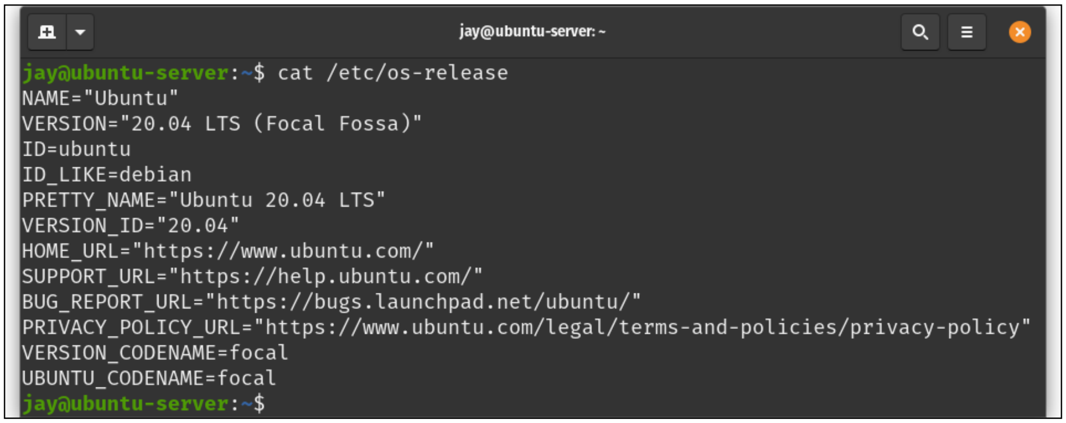
We can print the contents of a file to the screen with the `cat` command, along with a filename as an argument. The following command, for example, can be used to view the contents of `testfile.txt` in our current working directory, which we created earlier in the chapter when we discussed the `touch` command. Sure, this exercise is rather pointless since that file is empty, but it offers a good first example:

```
cat testfile.txt
```

There's no output, since again, the file is empty. So, let's instead take a look at a more practical example. Here's a file we can use `cat` against that actually does have content inside:

```
cat /etc/os-release
```

The output of this command is as follows:



```
jay@ubuntu-server:~$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04 LTS"
VERSION_ID="20.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=focal
UBUNTU_CODENAME=focal
jay@ubuntu-server:~$
```

Figure 4.6: Viewing the contents of /etc/os-release

The `/etc/os-release` file is one that exists on many distributions. It's a special file that gives you some information about the distribution that's currently installed. If you were to connect to a Linux server and wanted to know which distribution it was running, viewing the contents of this file is one way to find out. You can also view some of the same info in abbreviated form via the following command:

```
lsb_release -a
```

That command also works on various distributions, but I prefer the `/etc/os-release` file because it contains more information. Regardless, the entire point of this exercise is to demonstrate that the `cat` command allows you to view the contents of a file. More or less, there are other commands that also let you do the same thing. And I mean that literally—you can also try viewing the contents of `/etc/os-release` with `more` or `less`:

```
more /etc/os-release
less /etc/os-release
```

The `more` command allows you to more easily view larger files, so the benefit won't be as immediately apparent with a file as short as `/etc/os-release`. If you use `more` to view a longer file, it will stop the output when it fills the screen, and allow you to press *Enter* to advance to the next line.

The `less` command allows you to do the same thing, but also allows you to not only use your arrow keys in addition to *Enter* to view more output; it also allows you to advance forward or backward as well. So essentially, the `less` command gives you more features than the `more` command does.

In addition, we also have the `grep` command at our disposal. It's not typically used to simply view the contents of a file, but it's definitely a great command to know that can help you view specifically what you want to view, rather than the entire file.

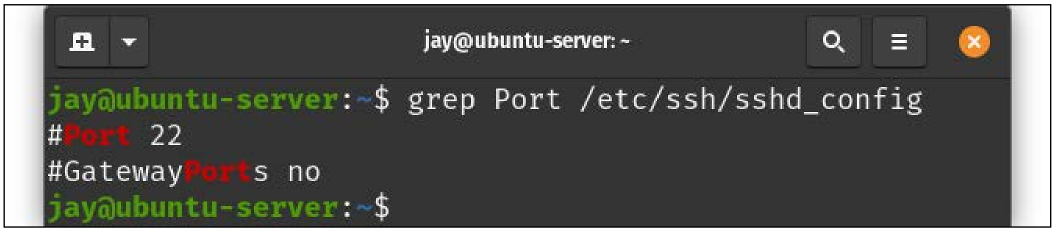
If you chose to install the OpenSSH server when you first installed Ubuntu Server, you should have the config file for it in your installation, and you can view the contents of that file with `cat` as we normally would:

```
cat /etc/ssh/sshd_config
```

Of course, that's going to dump the contents of that file onto our screen, and the file is many more lines than what the typical monitor is able to display all at once. We might be interested in a particular line or word, so we'll need to be able to narrow down the file to what we actually care about. We'll talk about the OpenSSH server in more detail in *Chapter 10*, *Connecting to Networks*, so don't worry about what this configuration file means yet. Let's just say, hypothetically, we're only interested in the port that OpenSSH is listening on. We can use the `grep` command to try and print only the lines of the `/etc/ssh/sshd_config` file that pertain to that specific configuration:

```
grep Port /etc/ssh/sshd_config
```

This command will produce the following output:



```
jay@ubuntu-server:~$ grep Port /etc/ssh/sshd_config
#Port 22
#GatewayPorts no
jay@ubuntu-server:~$
```

Figure 4.7: Viewing the contents of the /etc/ssh/sshd_config file with grep to find lines containing "Port"

Essentially, what we're doing is instructing `grep` to print only the lines of the `/etc/ssh/sshd_config` file that contain the string `Port`. In the screenshot, two lines contain a match for that string, so they're displayed. The first line is the one that's relevant to this example, so we didn't need to see the second line. But that output is certainly better than scrolling through the 124 lines of text in the file, when we're only interested in lines pertaining to the port.

By default, grep is case-sensitive. This means that if we were to use grep to find lines matching "port" (with a lowercase P), we would get no output at all. We can simply add the -i argument to make the search case-insensitive.

It's very common to see the grep command paired with another command, such as cat:

```
cat /etc/ssh/sshd_config | grep Port
```

That's a perfectly valid command, and will do the same thing. However, it's fairly redundant. We'll get into the concept of redirecting output in the next chapter, but essentially this command takes the output of the cat command and redirects it as input to the grep command. Using cat to first print out the file and then have grep grab the contents of that file and search it for a string is a two-step process where only one step is required. But again, it's still a valid command. To this day, I'll personally use the cat command with grep to do the same thing out of habit, as this was how all new Linux users were taught back when I started. You'll even see me do this in my YouTube videos—old habits are hard to break!

Log files are a great source of information about what's going on in the background of your server, and are especially useful for troubleshooting. We'll now transition into a discussion on viewing these files and some relevant commands pertaining to logging.

## Viewing application log files

In the last section of this chapter, let's explore log files a bit, as they bring several concepts that we've discussed full-circle. We went through an overview of the default directory layout, practiced viewing files, and we learned how to search files for strings. We'll discuss log files in greater detail later on in the book, but we can use all of these concepts to take an initial look at viewing log files now.

If you recall, during the discussion of the Linux filesystem layout earlier in this chapter, there was a table showing some of the most common directories that exist. Among the items in that table, I called out the /var/log directory. While logging is transitioning to a different style (more on that in *Chapter 22*, *Troubleshooting Ubuntu Servers*), we'll have a series of logs in the /var/log directory. Go ahead and use the ls command yourself, and you'll see there are quite a few files there. While I won't go through all of them in this chapter, let's take a look at /var/log/syslog.

The content of this file is going to have quite a few lines. This is the system log, which is used to view informational messages about what's going on in the background as Ubuntu runs on your server, and will show warnings and errors as well. If you run into a problem where something isn't working quite right, you may see output in the system log that will provide you with some sort of error you can look up in a search engine so you can try and find a resolution. For example, you can use `grep` to view any lines that contain the search term *Network* if you're having issues with your connection:

```
grep Network /var/log/syslog
```

That's just a hypothetical example, but it may show lines that are relevant. Adjust your search term to whatever you're interested in finding.

This is also a good time to introduce you to the `head` and `tail` commands. These commands will show the first ten or last ten lines of a file, respectively. This is useful for the `/var/log/syslog` file because again, that file is quite large, and you may only be interested in certain lines. You can also adjust the number of lines the `head` and `tail` commands show with the `-n` option with a desired number of lines. For example, to view the last `100` lines of a file:

```
tail -n 100 /var/log/syslog
```

Perhaps even more useful, is the `-f` option:

```
tail -f /var/log/syslog
```

This allows you to follow (watch) a file in (almost) real time. With the `-f` option, the terminal will continue to display new lines in this file as they're appended, so you can monitor the log file as someone attempts to reproduce a problem, for example. You can press *Ctrl* + *c* on your keyboard to break out of the follow mode and return back to the command prompt.

Of course, there are many more foundational commands and concepts that we can go over, but I think that this is enough for now. In the next chapter, we'll expand on this further. But for now, I recommend you practice all of the concepts in this chapter until you're familiar with them before we go on.

## Summary

There are more Linux commands than you'll ever be able to memorize. Most of us just memorize our favorite commands and variations of commands, and you'll develop your own menu of these commands as you learn and expand your knowledge. In this chapter, we covered many of the foundational commands that are, for the most part, essential. Commands such as `grep`, `cat`, `cd`, `ls`, and others were explored this time around. The next chapter is essentially a continuation of this one, but I wanted to split the foundational concepts into two chapters rather than one enormous one.

In the next chapter, we'll expand our foundational knowledge with a deeper look at file management, including editing files, input/output streams, symbolic links, and we'll even reveal the secret to life itself. Well, maybe not the latter, but the next chapter will still be great. See you there!

## Further reading

- Filesystem Hierarchy Standard: `http://www.pathname.com/fhs/`
- UsingTheTerminal: `https://help.ubuntu.com/community/UsingTheTerminal`
- Linux Commands for Beginners (Part 4) Navigating the Filesystem: `https://learnlinux.link/lcb_filesystem`

# 5
# Managing Files and Directories

In *Chapter 4*, *Navigating and Essential Commands*, we started looking deeper into Linux commands. We went over the most essential commands, covered the filesystem layout, as well as various methods to view the contents of files. In this chapter (as well as the next), we'll continue to expand on the command line and become more efficient while using the terminal. This time around, we'll expand a bit further on file management, take a look at input/output streams, and we'll also develop an understanding of symbolic links. Along the way, we will cover:

- Copying, moving, and renaming files and directories
- Editing files with the Nano and Vim text editors
- Input and output streams
- Using symbolic and hard links

Let's begin the chapter by taking a look at some methods we can use to alter the files and directories in the filesystem, such as copying and moving them.

## Copying, moving, and renaming files and directories

At this point, you should know how to move around within the filesystem (commands such as `cd`), inspect the contents of directories (`ls`), and even how to create empty files (the `touch` command). We even know how to remove files as well, such as executing the `rm` command against a file or directory. But until now, we haven't looked at moving files around within your Ubuntu filesystem.

First, to make a copy of a file or directory, we use the `cp` command. Copying a file is fairly easy, and such a command would look similar to the following:

```
cp file1 file2
```

In that example, `file2` is created as an exact copy of `file1`. Copying a file is useful in many situations, some of the most common of which are:

- Copying a file to a backup medium, such as an external drive or network share
- Creating a copy of a file before making a change, such as before editing a very important text file
- Duplicating a log file for a point-in-time analysis

Let's look at the last bullet point as another good example. We can capture a copy of the system log and store it in our current working directory by executing the following command:

```
sudo cp /var/log/syslog /home/<username>/syslog
```

The `cp` command is fairly simple: we give it a path to a file to copy, and then we type the path to the destination and desired filename. We also use `sudo` due to the fact that the `syslog` file is not readable by normal users by default. In the case of this command, a copy of the original `syslog` file will be saved in your current working directory.

For this particular example, log files are constantly being written to. It can sometimes be hard to troubleshoot an issue that occurred during a certain time if the file is continually expanding. That's not the only reason though. We certainly wouldn't want to make an accidental change to the log file and risk contaminating it or losing important information.

The previous command can actually be simplified a bit:

```
sudo cp /var/log/syslog .
```

In that example, we removed the target path and name, and replaced it with a period. The reason this works is that the period represents our current working directory. This isn't specific to the `cp` command either. There actually is a directory named with a period in every directory, which is essentially a pointer to the current directory. Therefore, if you're already inside the directory where you want the file to be copied to, you don't have to type the path. You don't have to type the name either, if you want the name of the file to be the same as the source.

Copying files (as well as moving them, which we'll look at next) is potentially destructive. If the target path and name already exist, then the target will be overwritten. By default, you won't see a confirmation before the target file is overwritten. As with all commands, take care that you really want to do whatever it is you're telling the command-line interpreter to do.

When it comes to copying directories, the `cp` command by itself won't do the trick:

```
sudo cp /var/log/apt .
```

The `/var/log/apt` directory contains log files that keep track of transactions performed with the `apt` command. It can be useful to keep an eye on what your other admins are installing. When it comes to this example though, the previous command will fail with the following error:

```
cp: -r not specified; omitting directory '/var/log/apt'
```

The error literally tells you what to do about it, but essentially the error is telling you that directories are omitted by default. In order to copy directories, you'll need to include the `-r` option. This stands for *recursive*, which is an option you'll see as a possibility for a handful of Linux commands. It tells the command-line interpreter to not only grab the object at the path you specified, but to do so recursively and include child objects as well. Therefore, the following command will work:

```
sudo cp -r /var/log/apt .
```

With that command, a local copy of the `/var/log/apt` directory will be stored in your current working directory, along with the contents.

When it comes to moving a file or directory from one place to another, we use the `mv` command. The syntax is almost exactly the same as the `cp` command. The difference is that instead of copying a file or directory, we're moving it. In that regard, it's probably self-explanatory how it works. Consider the following example commands:

```
mv file1 /path/to/new/directory/file1
mv file1 file2
```

With the first example, we're assuming that `file1` is in our current working directory. We're grabbing that file and moving it to `/path/to/new/directory` and giving it the same name of `file1` in that new directory. Just like with the `cp` command, we could've omitted the filename at the target, since it's staying the same. If the file already exists at the target directory with that name, it will be overwritten. So, the `mv` command is also potentially destructive, but more so when compared to `cp` since you are moving, instead of copying.

The second `mv` command is a bit more interesting, because in that example we're renaming a file. In Linux, there's no dedicated rename command, so the `mv` command is used for that purpose. In fact, `mv` is somewhat of a Swiss army knife due to the fact it serves multiple purposes. With it, you can move a file or directory, rename it, and also move a file to overwrite another file in a different location. It all depends on the source and destination paths. If the target exists, it will be overwritten. If not, the file will be renamed and/or moved to that path.

When managing files, you'll certainly come to a situation where you need to manage multiple files or directories. It's worth noting that the `cp` command, as well as the `mv` command, can be used with multiple objects at once; for example, if you have three directories, such as `dir1`, `dir2`, and `dir3`, and you need to move them into a new sub-directory. You could execute three `mv` commands to move each one separately, but you can also move all three with a single `mv` command:

```
mv dir1 dir2 dir3 /path/to/new/location
```

The same holds true with `cp` and with files; both `cp` and `mv` are set up to allow you to move or copy multiple directories or files with a single command.

Now that we know how to move files around, we should also take a look at how to edit them. There are many text editors available to us on the Ubuntu platform, with Nano and Vim being among the most common. In the next section, we'll go over the basics of both.

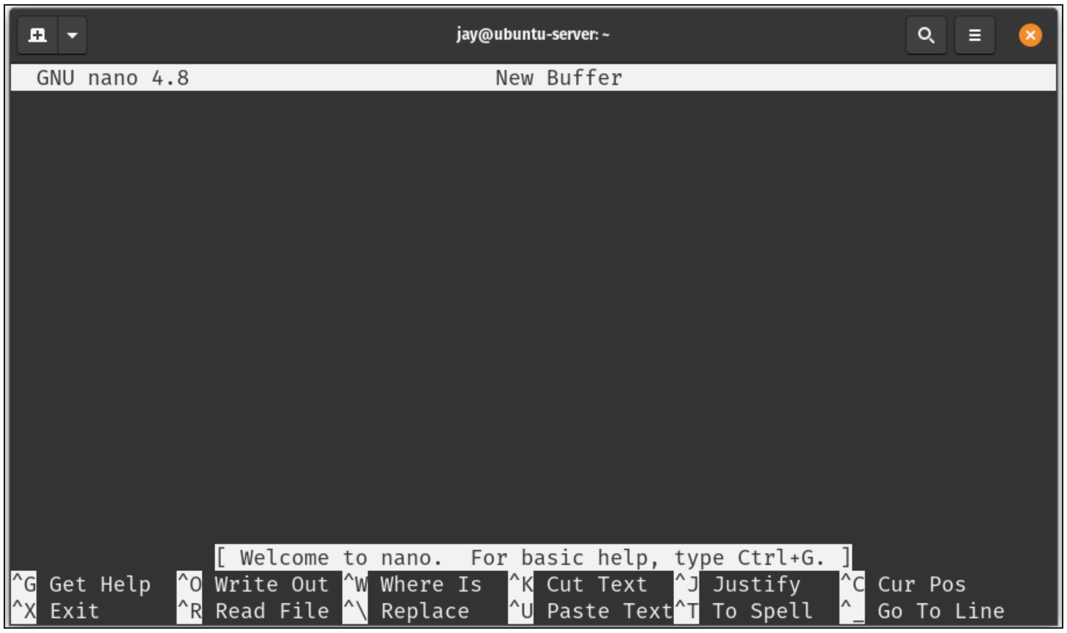
## Editing files with the Nano and Vim text editors

Now that we know how to copy and move files, it would be useful to know how to edit them. There are multiple forms of text editors for Ubuntu, some available in the command line, and others in graphical environments such as gedit in the desktop version.

Some may feel that command-line text editors are more complicated than **Graphical User Interface** (**GUI**) editors (and to be honest, they can be), but the main benefit is that you can use the same editor regardless of whether or not you have a GUI. In a way, this means the non-graphical editors are a bit more portable, and you can rely on them more. Almost all installations of Ubuntu will include the `nano` text editor, and you can rely on that more often than something like `gedit` being available. In addition, the `vim` editor is another popular consideration. It's a bit more advanced than `nano`, but in my opinion, much more powerful. In the following sections, we'll look at both `nano` and `vim`.

## Editing with Nano

The `nano` editor, while more basic in terms of features, still has a fairly strong following. Actually, if you haven't already noticed, a person's choice of editor can actually be somewhat of a debate in the Linux community. To launch the `nano` editor, it's as simple as running the `nano` command. If you don't provide `nano` with a filename, it simply starts with an empty window, as shown in the following screenshot:



```
  GNU nano 4.8                    New Buffer




                [ Welcome to nano.  For basic help, type Ctrl+G. ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Paste Text ^T To Spell   ^_ Go To Line
```

Figure 5.1: The nano text editor, with no file selected

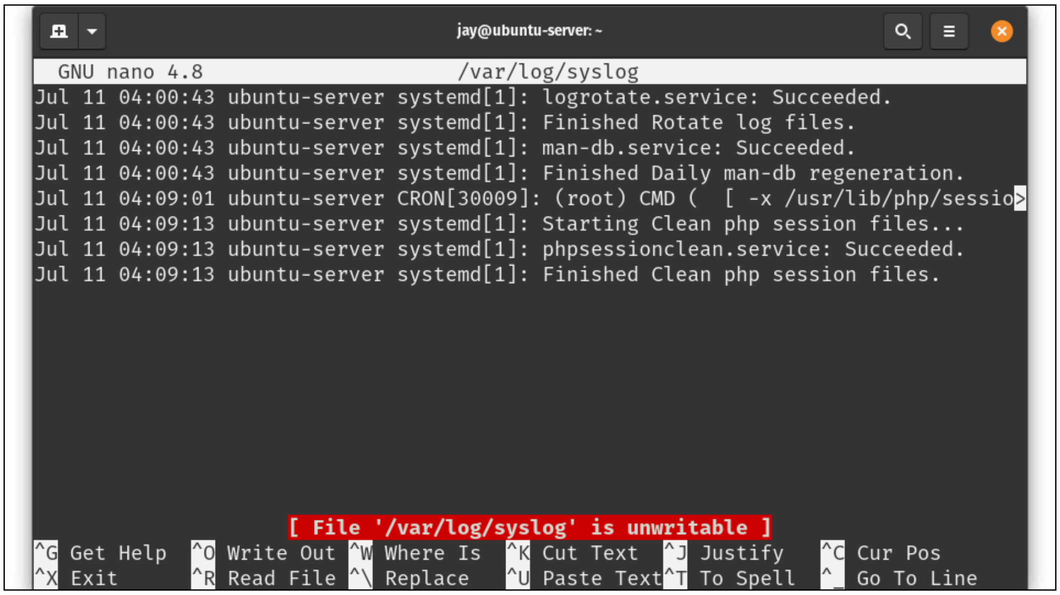
You can start typing right away, and when you want to save the file, you can simply press *Ctrl* + *o* to do so. On the bottom of the nano screen, you'll see an overview of shortcuts, including the method of saving the file I just mentioned (which it refers to as **Write Out**). As you can see then, *Ctrl* + *x* is how you exit the editor and go back to the command line.

One trick that I love (which also works well with vim) is pressing *Ctrl* + *z* when you have nano open, which will make it disappear. Actually, not really. That keyboard shortcut sends the current application inside the terminal to the background. Effectively, it's the same as minimizing it. You can bring the nano window back by executing fg, which is short for *foreground*. The ability to background and foreground an application in the terminal is part of process management, which is actually something we'll talk about in *Chapter 7, Controlling and Managing Processes*, but we're giving you this tip ahead of time, to help you on your way! When it comes to editing files, sometimes it makes sense to send the editor to the background and bring it back later.

As you can probably guess, you can also use the nano command, with a target path and filename, to have the editor come up with a file already open. For example:

```
sudo nano /var/log/syslog
```

This command will result in the file opening in nano:



```
  GNU nano 4.8                    /var/log/syslog
Jul 11 04:00:43 ubuntu-server systemd[1]: logrotate.service: Succeeded.
Jul 11 04:00:43 ubuntu-server systemd[1]: Finished Rotate log files.
Jul 11 04:00:43 ubuntu-server systemd[1]: man-db.service: Succeeded.
Jul 11 04:00:43 ubuntu-server systemd[1]: Finished Daily man-db regeneration.
Jul 11 04:09:01 ubuntu-server CRON[30009]: (root) CMD (  [ -x /usr/lib/php/sessio>
Jul 11 04:09:13 ubuntu-server systemd[1]: Starting Clean php session files...
Jul 11 04:09:13 ubuntu-server systemd[1]: phpsessionclean.service: Succeeded.
Jul 11 04:09:13 ubuntu-server systemd[1]: Finished Clean php session files.

                    [ File '/var/log/syslog' is unwritable ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Paste Text^T To Spell    ^_ Go To Line
```

Figure 5.2: The nano text editor, with the syslog file open

To be fair, editing a log file directly is probably not a good idea in practice, but the example works. The syslog is a file you may want to open and inspect, but it would be a good idea to check the size of the syslog file with the `ls -lh /var/log/syslog` command to make sure it's not excessively large in size, which can slow down the server.

All of that aside, the takeaway here is that you can open a file with `nano` directly by simply providing it with a path to something. Inside the editor window, you can move around with the arrow keys, just like you would in a graphical editor. You can also search for a specific string of text, by pressing *Ctrl* + *w* for **Where Is**, as it mentions in the list of actions near the bottom.

In addition, note the text in the screenshot that is highlighted in red. It's simply telling us that we don't have access to edit the file. In that example, I didn't use `sudo` because I didn't intend on making any changes to the file. I was able to open it without `sudo` since my user is a member of the `syslog` group. Sometimes, I'll open files in an editor that I don't have write permission to make changes to, which serves as a fail-safe in case I accidentally do make a change. To make a real change, I can close the file and reopen it with `sudo`.

Anyway, the `nano` editor is fairly simplistic. Is there more to it than that? Sure, but the most important thing to know is how to open files, edit them, and save them, which we've covered in this section. Of course, you can practice with the actions that are presented to you at the bottom of the window to go beyond normal usage.

Now, let's take a look at Vim.

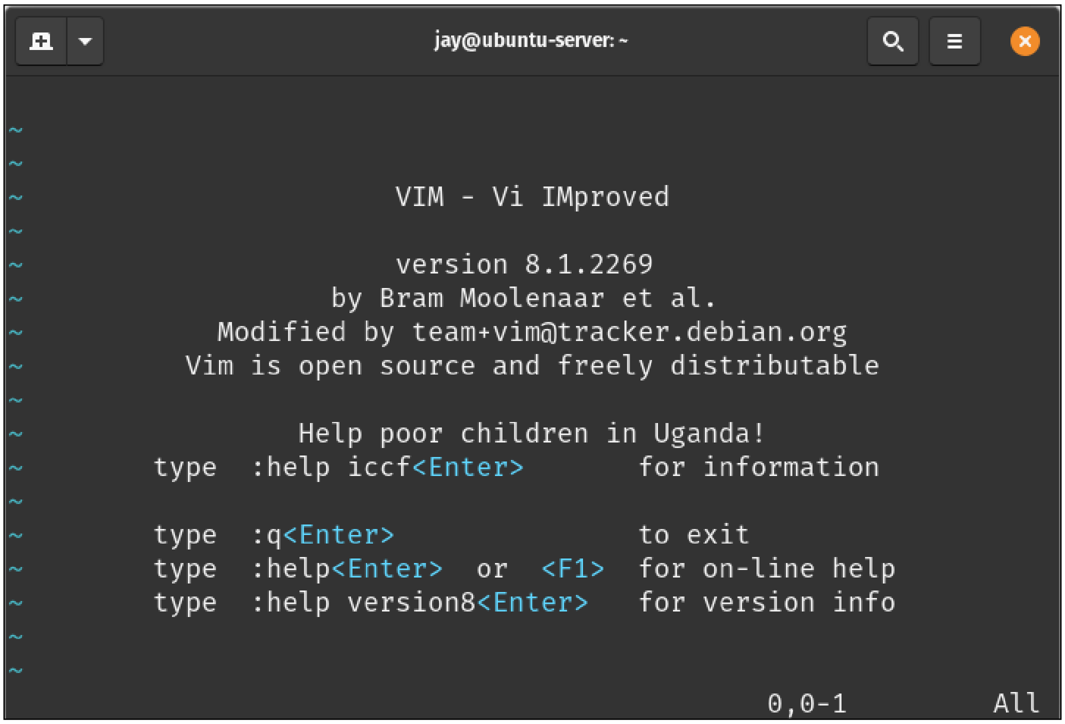
## Editing with Vim

Vim is my favorite editor, and the one I use most often. It's a little on the advanced side, but it's not to the point where it's frustrating. By default, it's not actually installed. Personally, I prefer the `vim-nox` package, which can be installed by executing the following command:

```
sudo apt install vim-nox
```

It really doesn't matter which of the variations of `vim` you install. Each variation adds its own features. In the case of `vim-nox`, it has built-in support for scripting languages, but is not really all that different from standard `vim` in other regards. The concepts you'll learn here are not specific to this version.

Just like `nano`, the `vim` editor can be called by itself by entering the command `vim` with no options, or with a path to a file, such as `vim /home/myuser/myfile.txt`. With no file chosen, `vim` will display its default help text. It gives you some of the default commands, such as `:q` for quitting the editor:



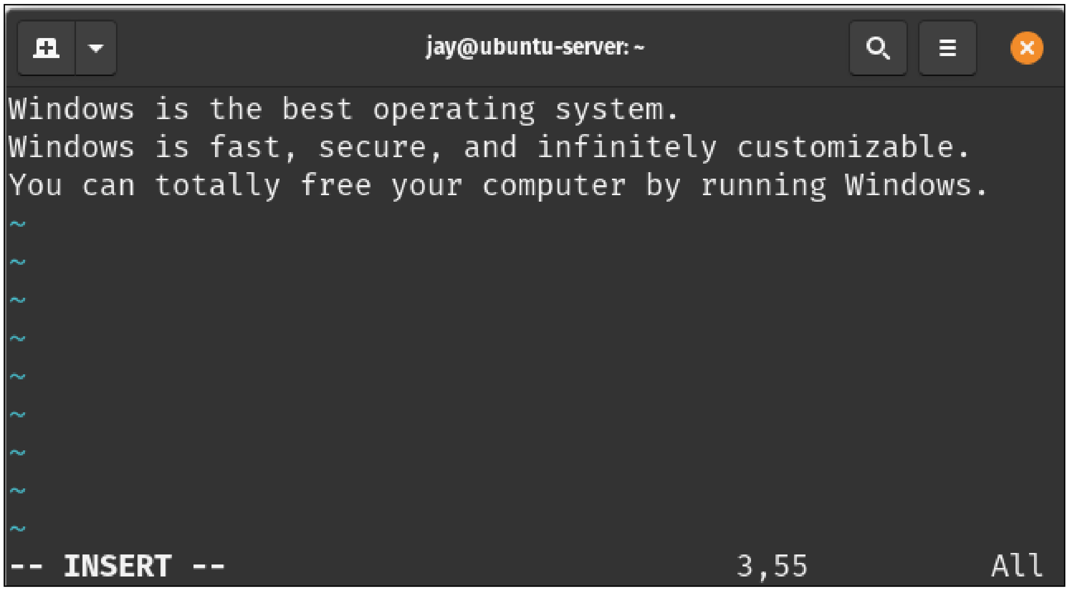
```
jay@ubuntu-server: ~

~
~
~                          VIM - Vi IMproved
~
~                          version 8.1.2269
~                      by Bram Moolenaar et al.
~               Modified by team+vim@tracker.debian.org
~             Vim is open source and freely distributable
~
~                     Help poor children in Uganda!
~          type  :help iccf<Enter>       for information
~
~          type  :q<Enter>               to exit
~          type  :help<Enter>  or  <F1>  for on-line help
~          type  :help version8<Enter>   for version info
~
~
                                                  0,0-1         All
```

Figure 5.3: The vim text editor, with no file open

When you start `vim`, you start in *command mode*, which is one of several different modes the editor can be in. In command mode, you can't actually edit text. Command mode allows you to run commands, as the name implies, which can allow you to manipulate text in really neat ways.

If you want to edit a file as you were already doing in `nano`, you'll need to switch to *insert mode* to do so. You can switch to insert mode by pressing the *Insert* key on your keyboard, or the *i* letter key. Once you're in insert mode, you can start typing as you normally would in any other editor. You'll move the cursor around with the arrow keys, and insert text wherever you want. You can also navigate with the *h*, *j*, *k*, and *l* keys in place of the arrow keys (which some users actually prefer). To exit insert mode, you can press *Esc*. That will bring you back to command mode.

At first, the different modes of `vim` can be a bit confusing to newcomers. For me, I see it as somewhat of a superpower. With `vim`, you have a mode for editing text, and another dedicated mode for manipulating it. When it comes to text manipulation, that's one of the strongest features of `vim`. For an example of this, consider the following hypothetical file:



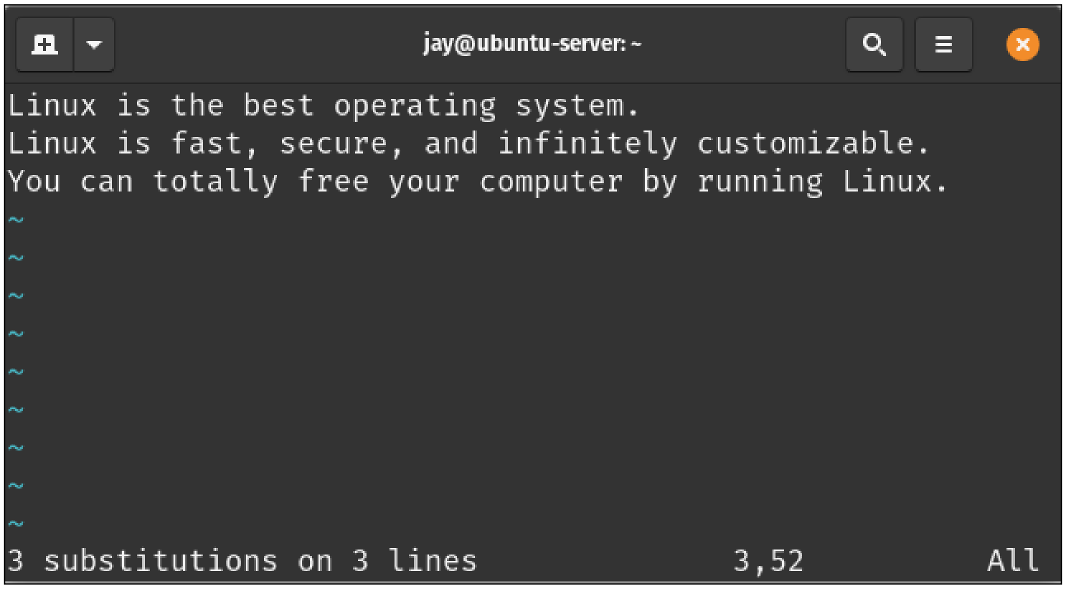
```
Windows is the best operating system.
Windows is fast, secure, and infinitely customizable.
You can totally free your computer by running Windows.
~
~
~
~
~
~
~
~
~
-- INSERT --                             3,55          All
```

Figure 5.4: The vim text editor, with a sample file

I don't know why, but something about that file seems off to me. We'll need to make a very important correction to try to fix it. First, you can see from the screenshot that the editor was left in insert mode. To do some text manipulation, we first need to press *Esc* to go back to command mode. Once there, we can get ready to type a command. And the command we want to type is this one:

```
:%s/Windows/Linux/g
```

I don't know about you, but personally I think the file looks better now after running that command:



```
Linux is the best operating system.
Linux is fast, secure, and infinitely customizable.
You can totally free your computer by running Linux.
~
~
~
~
~
~
~
~
~
3 substitutions on 3 lines                      3,52          All
```

Figure 5.5: The vim text editor, after running a sample command

Technically, Linux is a kernel, and not an operating system. Ubuntu, a distribution of Linux, is the closest equivalent of an operating system for us. But I'm willing to let that slide, for now. Other than that, in my opinion, this version of the file is more accurate!

This was a more advanced example of `vim` usage that would normally be outside the scope of this book. I wanted to show you an example of the power of this amazing editor, and finding and replacing text is just one of the many things we can do with it. Some people (including myself) even go as far as to install plugins to essentially turn `vim` into an **Integrated Development Environment** (**IDE**). In the example we just used, we were able to find and replace all occurrences of a string with another string. This was also an example of the types of tasks we carry out in command mode, which allows us to manipulate text and enter more advanced commands. With this admittedly contrived example, you can immediately see the value of having multiple modes in `vim`.

I've been using vim for about 8 years or so, because I can never seem to figure out how to exit it. Okay, to be fair, that joke is getting to be a bit old now. If you've already been using Ubuntu or any other Linux distribution for a while, that joke probably made you cringe a little. If that was the first time you've seen it, then I'm glad I was the one that introduced you to it—for some reason, vim has a reputation of being difficult to exit. But that's not really true at all, it's actually quite easy. In command mode, you simply enter :q to exit the editor:

```
:q
```

If you've made changes, the :q command won't let you exit, but you can force exit if you add an exclamation mark, so the command becomes :q!. If you'd also like to save changes while exiting, add w, so the command inside vim would become:

```
:wq
```

Essentially, we're exiting the editor (q) and writing the file at the same time (w). In command mode, commands start with a colon (:) and then the actual command. There are so many commands available that it's impossible to talk about them all in one chapter. What I'll do, then, is mention some that I consider to be the most important.

First, it's actually possible to run a shell command without leaving vim by using the following command:

```
:! <shell command>
```

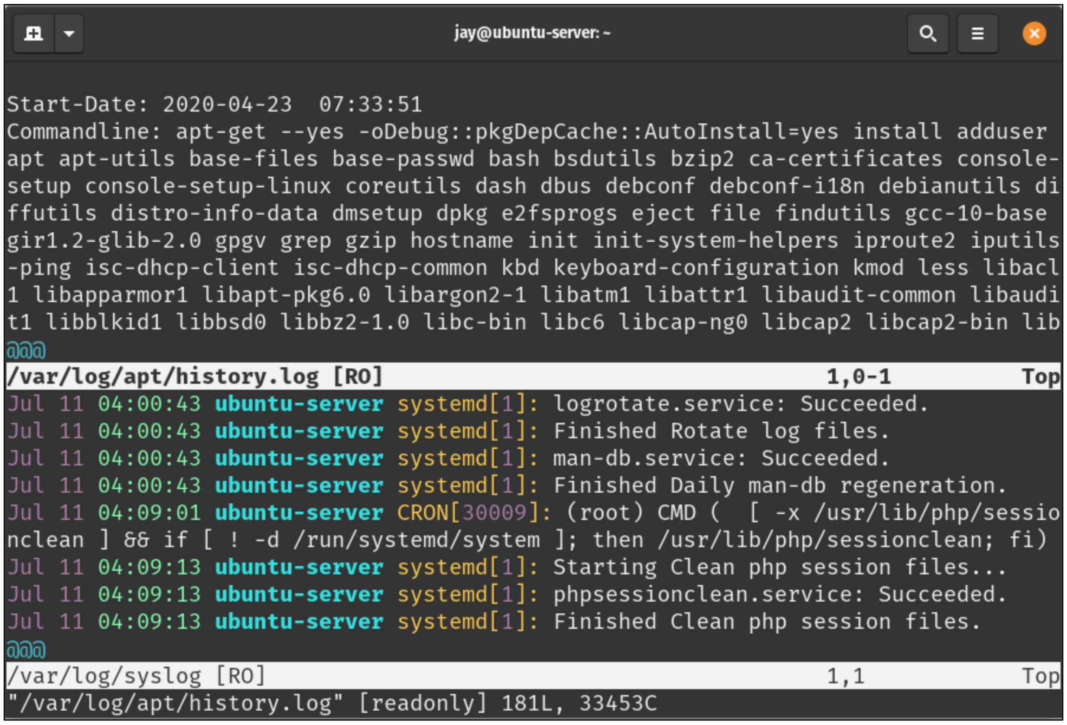
The exclamation mark allows you to run a command, and then you type the actual command. For example:

```
:! ls -l /var/log
```

The previous example will show you the contents of the /var/log directory, and then you can press *Enter* to return back to vim. The :sp command is short for *split*:

```
:sp /path/to/file
```

In this case, `vim` is able to show you more than one file at a time in the same window, effectively splitting the window to show you both files. With `:split`, or `:sp` for short (both do the same thing), it will either split the file into two views (the same file open in each) or it will show a separate file in the other split, if you give it a filename. This command will split the file horizontally:



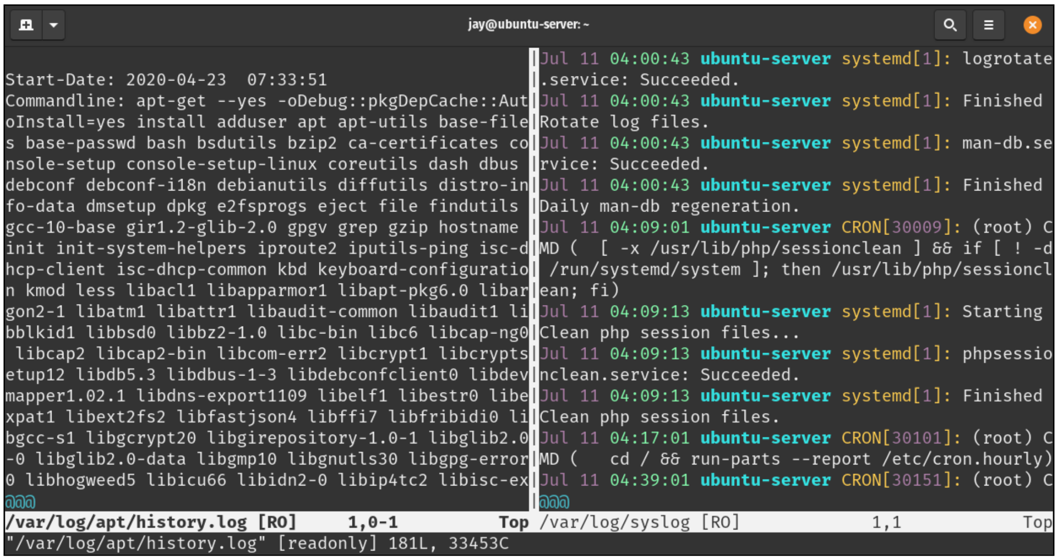
```
Start-Date: 2020-04-23  07:33:51
Commandline: apt-get --yes -oDebug::pkgDepCache::AutoInstall=yes install adduser
apt apt-utils base-files base-passwd bash bsdutils bzip2 ca-certificates console-
setup console-setup-linux coreutils dash dbus debconf debconf-i18n debianutils di
ffutils distro-info-data dmsetup dpkg e2fsprogs eject file findutils gcc-10-base
gir1.2-glib-2.0 gpgv grep gzip hostname init init-system-helpers iproute2 iputils
-ping isc-dhcp-client isc-dhcp-common kbd keyboard-configuration kmod less libacl
1 libapparmor1 libapt-pkg6.0 libargon2-1 libatm1 libattr1 libaudit-common libaudi
t1 libblkid1 libbsd0 libbz2-1.0 libc-bin libc6 libcap-ng0 libcap2 libcap2-bin lib
@@@
/var/log/apt/history.log [RO]                                  1,0-1          Top
Jul 11 04:00:43 ubuntu-server systemd[1]: logrotate.service: Succeeded.
Jul 11 04:00:43 ubuntu-server systemd[1]: Finished Rotate log files.
Jul 11 04:00:43 ubuntu-server systemd[1]: man-db.service: Succeeded.
Jul 11 04:00:43 ubuntu-server systemd[1]: Finished Daily man-db regeneration.
Jul 11 04:09:01 ubuntu-server CRON[30009]: (root) CMD (   [ -x /usr/lib/php/sessio
nclean ] && if [ ! -d /run/systemd/system ]; then /usr/lib/php/sessionclean; fi)
Jul 11 04:09:13 ubuntu-server systemd[1]: Starting Clean php session files...
Jul 11 04:09:13 ubuntu-server systemd[1]: phpsessionclean.service: Succeeded.
Jul 11 04:09:13 ubuntu-server systemd[1]: Finished Clean php session files.
@@@
/var/log/syslog [RO]                                           1,1            Top
"/var/log/apt/history.log" [readonly] 181L, 33453C
```

Figure 5.6: The vim text editor, with two files open in the same window

The previous screenshot shows two files open, `/var/log/apt/history.log` and `/var/log/syslog`. You may also notice that the status bar for each file, at the bottom, shows `RO`. As you can probably guess, this stands for *read only* and displays on the screen because I'm viewing files that only `root` has access to change, and I didn't use `sudo`.

To switch between the two files, we can press the *w* key twice while holding *Ctrl*. When we do that, our insertion point moves from one split to the other. To exit each individual buffer (or the editor itself, if we only have one file open), we can press *Esc* to return to command mode, and then enter the `:q` or `:q!` command to exit without saving changes, or `:wq` to exit while also saving changes, just like before.

An alternative to `:split` is `:vsplit`, or `:vs` for short. It does the same thing as `:split`, but it splits the window vertically. Considering most computer displays nowadays are widescreen (and even ultra-widescreen), a vertical split often works out better in a practical sense:



```
jay@ubuntu-server: ~
                                                    |Jul 11 04:00:43 ubuntu-server systemd[1]: logrotate
Start-Date: 2020-04-23  07:33:51                    |.service: Succeeded.
Commandline: apt-get --yes -oDebug::pkgDepCache::Aut|Jul 11 04:00:43 ubuntu-server systemd[1]: Finished
oInstall=yes install adduser apt apt-utils base-file|Rotate log files.
s base-passwd bash bsdutils bzip2 ca-certificates co|Jul 11 04:00:43 ubuntu-server systemd[1]: man-db.se
nsole-setup console-setup-linux coreutils dash dbus |rvice: Succeeded.
debconf debconf-i18n debianutils diffutils distro-in|Jul 11 04:00:43 ubuntu-server systemd[1]: Finished
fo-data dmsetup dpkg e2fsprogs eject file findutils |Daily man-db regeneration.
gcc-10-base gir1.2-glib-2.0 gpgv grep gzip hostname |Jul 11 04:09:01 ubuntu-server CRON[30009]: (root) C
init init-system-helpers iproute2 iputils-ping isc-d|MD (   [ -x /usr/lib/php/sessionclean ] && if [ ! -d
hcp-client isc-dhcp-common kbd keyboard-configuratio| /run/systemd/system ]; then /usr/lib/php/sessioncl
n kmod less libacl1 libapparmor1 libapt-pkg6.0 libar|ean; fi)
gon2-1 libatm1 libattr1 libaudit-common libaudit1 li|Jul 11 04:09:13 ubuntu-server systemd[1]: Starting
bblkid1 libbsd0 libbz2-1.0 libc-bin libc6 libcap-ng0|Clean php session files...
 libcap2 libcap2-bin libcom-err2 libcrypt1 libcrypts|Jul 11 04:09:13 ubuntu-server systemd[1]: phpsessio
etup12 libdb5.3 libdbus-1-3 libdebconfclient0 libdev|nclean.service: Succeeded.
mapper1.02.1 libdns-export1109 libelf1 libestr0 libe|Jul 11 04:09:13 ubuntu-server systemd[1]: Finished
xpat1 libext2fs2 libfastjson4 libffi7 libfribidi0 li|Clean php session files.
bgcc-s1 libgcrypt20 libgirepository-1.0-1 libglib2.0|Jul 11 04:17:01 ubuntu-server CRON[30101]: (root) C
-0 libglib2.0-data libgmp10 libgnutls30 libgpg-error|MD (   cd / && run-parts --report /etc/cron.hourly)
0 libhogweed5 libicu66 libidn2-0 libip4tc2 libisc-ex|Jul 11 04:39:01 ubuntu-server CRON[30151]: (root) C
@@@                                                 |@@@
/var/log/apt/history.log [RO]      1,0-1         Top /var/log/syslog [RO]                   1,1          Top
"/var/log/apt/history.log" [readonly] 181L, 33453C
```

Figure 5.7: The vim text editor, with two files open in a vertical split

To be honest, the effect doesn't look all that great in the previous screenshot, the reason being the width of a page in a book isn't enough to really show the benefit. Go ahead and give it a shot and see for yourself.

So far, we've gone over two modes of `vim`—command and insert mode. But there's also another commonly used mode we haven't discussed yet—*visual mode*. Visual mode allows you to select text, which then allows you to copy and paste.

To do so, make sure you're in command mode, and then start by moving the cursor to the first character in a series of text you want to copy. Then, press the *v* letter key on your keyboard, and move the selection with your arrow keys. You'll notice that you'll highlight more text as you move the cursor. Once you've selected all the text that you'd like to copy, press the *y* key and the highlight will go away. At this point, you've essentially copied the text you had highlighted into the `vim` equivalent of the clipboard, similar to desktop operating systems. You can then (while also in command mode) press the *p* key to paste the text wherever the cursor is. It may take a few tries to get a hang of the workflow, but go ahead and give it a shot. If you make a mistake, you can press *u* while in command mode to *undo*, so you can revert any changes you make.

Again, `vim` is a somewhat advanced editor. Learning the basic functions is fairly easy, so knowing how to open, edit, close, and save files is something you can learn fairly quickly. The feature set of `vim` is so vast that I'm still learning new tricks many years after I started using it. To make sure you understand the basics though, let's summarize the basic workflow of `vim` to help you memorize it. It looks something like this:

1. Open Vim by either typing `vim` by itself, or with a filename: `vim <filename>`.
2. You start in *command mode*. This mode is great for running `vim` commands. Press the *Insert* or the *i* keys to switch to *insert mode*.
3. While in insert mode, you can edit the text by typing with your keyboard and moving the cursor around with the arrow keys.
4. When you're done editing the file, press *Esc* to return to command mode.
5. To exit without saving changes, enter the `:q` or `:q!` command. If you do want to save changes, enter `:w` by itself to save and remain in the editor, or `:wq` to save and quit `vim` at the same time.

Entire books can be written about `vim` (and there have been), so it would be outside the scope of this book to go into exhaustive detail. In fact, I even have a dedicated video tutorial series about it at `www.learnlinux.tv` if you're interested in learning more. But for now, I'll leave you with a few more useful tips that I think you should know.

While in command mode, you can press the *x* key to delete a single character, wherever the cursor is. You can press the *d* letter key *twice* in succession to delete an entire line. When you do this, that line is also copied into the paste buffer; you can paste that line by pressing *p*.

In addition, I mentioned several times that you can switch to insert mode by pressing *i* or *Insert*. You can also do so by pressing the *a* letter key with or without holding *Shift*. If you don't hold *Shift*, you'll enter insert mode one character to the right. This isn't very useful to me, but typing a capital *A* is something I find myself doing a lot, which enters you into insert mode while also taking you to the very end of the line. Since I often want to start typing after the end of a sentence, this works well for me.

Also, while in command mode, you can type *Shift* + *g* for a capital *G* to immediately go to the very end of the file. Alternatively, you can press the *g* key *twice* in succession to move to the very top of the file.

Another trick I like is to turn on line numbers. This is useful, especially if an error message in a log file is complaining about a file regarding a specific line. The following command turns on line numbers:

```
:set number
```

The following command disables line numbers:

```
:set nonumber
```

If you'd always like to see line numbers by default, you can edit the `.vimrc` file inside your home directory. This file is read by `vim` each time you start it.

> Note that the `.vimrc` file begins with a period, which means that the file is hidden. Hidden files aren't normally shown to you when you use `ls` to list the contents of a directory. Using the `-a` option with the `ls` command will show all files, including the ones that are normally hidden.

Go ahead and open the `.vimrc` file in your editor:

```
vim ~/.vimrc
```

Most likely, this will be a blank file for you as a `.vimrc` file doesn't exist by default. There are many commands we can add to tweak the behavior of the `vim` editor, too many to mention in a single chapter. The following line is very useful though, as it ensures that line numbers are enabled by default:

```
set number
```

Leaving out the colon at the beginning of the line is intentional; it's not required inside the `.vimrc` file. From this point forward, each new `vim` session you open will start off with line numbers enabled. You can still disable line numbers as needed with the `:set nonumber` command within `vim` as we did earlier. In addition, you can add many more customizations to `vim` with the `.vimrc` file, but that's beyond the scope of this chapter. For now, just know that the possibility of customizing `vim` with the config file is something that you can do.

Now that we know how to edit files, we should also take a closer look at streams, which allow us to manipulate both input and output in various ways.

## Input and output streams

During our journey into Ubuntu Server so far, we've worked quite a bit within the terminal. We've been able to inspect the contents of files, insert text into files, and more. We've actually been working with **streams** the entire time without knowing it. In this section, we're going to talk about this subject in more detail.

If you've studied computer science at all, then you probably already know that *output* refers to things that are printed out of the computer (for example, text being printed to the screen, or onto paper from a printer) and *input* refers to data that is being entered into a computer, whether that be on the command line, into a file, or similar.

Linux takes this concept a bit further. Streams in Linux refer to a special way to handle what's going in or out, and beyond the input and output streams, we also have a third that refers to errors.

Output streams in Linux are referred to as **Standard Output**, and input streams are referred to as **Standard Input**. These are abbreviated as **stdout** and **stdin**, respectively. The reason for the expansion of the simple concept of input/output into a concept of its own is because, on the Linux shell, we can handle these streams differently and perform different tasks with them.

We've been working with standard output throughout the entire book so far. Everything that is printed to the terminal is standard output. For example, when you ran the `sudo apt install vim-nox` command earlier, the results of that command (the flood of text showing the status of the package installation as it occurred) was standard output. When you use the `cat /var/log/syslog` command to have the contents of `/var/log/syslog` dumped on to the screen, the contents being displayed are standard output. By far, standard output is something you'll work with the most.

To understand the concept of standard output better, let's take a look at redirection. The following command is an example:

```
cat /var/log/syslog > ~/logfile.txt
```

With that command, we're using `cat` to display the contents of `/var/log/syslog`. But instead of simply showing the contents on screen, we use the `>` character to redirect standard output into a file, `~/logfile.txt`. This means that standard output (showing the contents of the file) will not be shown at all, since we redirected that into a file. Similarly, we could have also run:

```
cat /var/log/syslog 1> ~/logfile.txt
```

Notice that I added a `1` right before the *greater-than* symbol. Standard output is designated by a file descriptor of 1. So, with that command, I'm specifically saying that I want to redirect standard output, and *only* standard output, into the file. Standard output is implied, so I didn't need to include the `1`. That's why simply using the *greater-than* symbol works for redirecting standard output to a file.

If you want to append a file rather than completely overwrite it, you can use two *greater-than* symbols (`>>`) to append, rather than overwrite. For example, the following command will add the contents of the `syslog` file to the end of the `logfile.txt` file, rather than overwrite the entire file:

```
cat /var/log/syslog >> ~/logfile.txt
```

Standard input also has a file descriptor, which is 0. Standard input is how commands receive data. Essentially, commands that accept input from the user do so by accepting `stdin`. Standard input is a bit more challenging to show in an example, but the following works:

```
cat /var/log/syslog | grep -i <keyword>
```

With that command, I'm grabbing the contents of the `/var/log/syslog` file and piping that into the `grep` command, which allows me to display only the lines that include a specific term. No two `syslog` files will be the same, since every server can be configured uniquely. But if you're attempting to investigate a specific application or service, you can `grep` the log for keywords you think might be appropriate. If you include the `-i` option, then `grep` will perform a search that's case-insensitive. In this example, the output of the `cat` command becomes standard input to the `grep` command. This command can be executed with any search term of your choice.

I could have also run:

```
cat < /var/log/syslog | grep Network
```

In that example, I'm using the *less-than* symbol to redirect the contents of `/var/log/syslog` to be standard input to the `cat` command. The `cat` command normally prints whatever text is fed to it to the screen as standard output, but instead, I'm using the pipe symbol, `|`, to grab that output and use it as standard input to the `grep` command.

This concept can be a bit confusing at first, but if you keep practicing it, it will definitely make sense. Let's look at another example, so we can understand standard error (**stderr**) as well:

```
find / -name "syslog"
```

The `find` command allows you to find files that match particular criteria, such as looking for files that are named `syslog` in this example. Here, we're searching the entire filesystem, because we used `/` to start the search from. The problem is, I don't have permission to read all the files on the filesystem, and I didn't use `sudo`. This is going to result in quite a few errors being printed to the screen, errors such as these:

```
find: '/var/lib/netdata/health': Permission denied
find: '/var/lib/netdata/registry': Permission denied
find: '/var/lib/netdata/cloud.d': Permission denied
find: '/var/lib/udisks2': Permission denied
```

Since the `find` command was used to search the entire filesystem, including places I don't have permission to look, our terminal will be flooded with errors. These errors are displayed using standard error, which has a file descriptor of 2. If we want to hide these errors, we can do this:

```
find / -name "syslog" 2> /dev/null
```

With that command, no errors will show up when we run it. That's because we instructed the interpreter to capture standard error and redirect it to `/dev/null`. `/dev/null` is a special device where things disappear forever. If you move or redirect something there, it's effectively deleted. Since a standard error has a file descriptor of 2, we combined that with the *greater-than* symbol to form `2>`, which basically instructs the shell to do a redirect, but to only redirect standard error and leave standard output alone. We can also choose to redirect more than one stream to different places within a single command:

```
find / -name "syslog" 1> stdout.txt 2> stderr.txt
```

With that variation, I'm redirecting successful output to `stdout.txt`, and the errors to `stderr.txt`. This allows us full control over where successful and unsuccessful messages are printed. This also helps us quite a bit when it comes to troubleshooting, because we may want to focus on only the errors, and getting rid of the success output may trim the number of lines we'll have to look through.

I recommend that you go ahead and practice this concept as much as you can; it's definitely something you'll want to commit to memory. You don't have to master this concept right now, but knowing the basics will provide a great foundation.

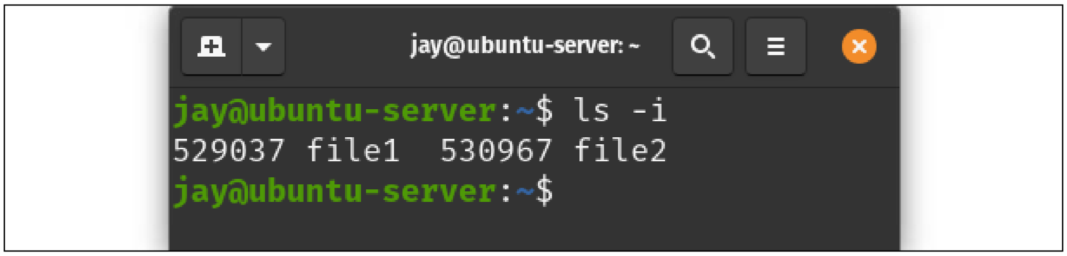
Next, let's discuss another aspect of file management—links. Sometimes, you'll need to link one thing to another, and there are a few ways to go about that along with some best-practice advice.

## Using symbolic and hard links

If you've used a graphical operating system for more than a week, you're probably more than familiar with the concept of shortcuts. Either on the desktop or within a menu, you will have shortcuts to files and applications. This can be a shortcut to your home or profile directory, a shortcut to an application, an individual file, and so on. We have the same concept in Linux.

With Linux, we can link files to other files, which gives us the ability to create our own shortcuts, which are effectively similar to shortcuts in graphical operating systems, but without the requirement of a GUI. This comes in the form of *symbolic* and *hard* links, which are two different methods in which we can link things. Symbolic and hard links are very similar, but to explain them, you'll first need to understand the concept of **inodes**.

An inode is a data object that contains metadata regarding files within your filesystem. Although a full walkthrough of the concept of inodes is beyond the scope of this book, think of an inode as a type of database object, containing metadata for the actual items you're storing on your disk. Information stored in inodes includes details such as the owner of the file, permissions, last modified date, and type (whether it is a directory or a file). Inodes are represented by an integer number, which you can view with the `-i` option of the `ls` command. On my system, I created two files: `file1` and `file2`. These files are inodes `529037` and `530967`, respectively. You can see this output in the following screenshot where I run the `ls -i` command. This information will come in handy shortly:



```
jay@ubuntu-server:~$ ls -i
529037 file1  530967 file2
jay@ubuntu-server:~$
```

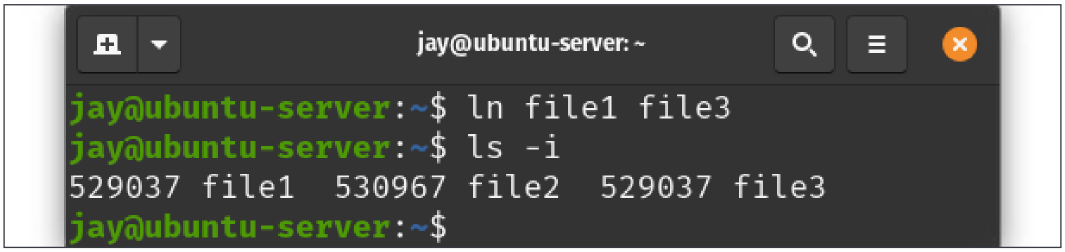
Figure 5.8: Output of ls -i

As I mentioned earlier, there are two types of links in Linux: symbolic links and hard links. While the two types of links approach the concept differently, they pretty much serve the same purpose. Basically, a link allows us to reference a file somewhere else on our filesystem.

For a practical example, let's create a hard link. In my case, I have a couple of files in a `test` directory, so I can create a link to any of them. To create a link, we'll use the `ln` command:

```
ln file1 file3
```

Here, I'm creating a hard link (`file3`) that is linked to `file1`. To play around with this, go ahead and create a link to a file on your system. If we use `ls` again with the `-i` option, we'll see something interesting:



```
jay@ubuntu-server:~$ ln file1 file3
jay@ubuntu-server:~$ ls -i
529037 file1  530967 file2  529037 file3
jay@ubuntu-server:~$
```

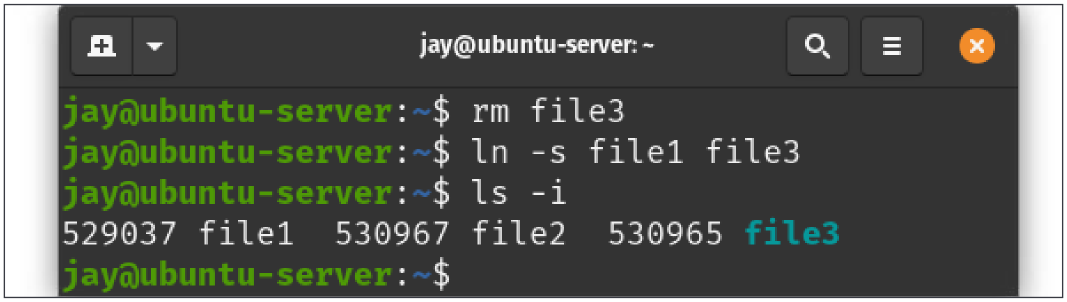
Figure 5.9: Output of second ls -I command

If you look closely at the output, both `file1` and `file3` have the same inode number. Essentially, a hard link is a duplicate directory entry, where both entries point to the same data. In this case, we created a hard link that points to another file. With this hard link created, we can move `file3` into another location on the filesystem and it will still be a link to `file1`. Hard links have a couple of limitations, however. First, you cannot create a hard link to a directory, only a file. Second, this link cannot be moved to a different filesystem. That makes sense, considering each filesystem has its own inodes. Inode `529037` on my system would, of course, not point to the same file on another system, if this inode number even exists at all.

To overcome these limitations, we can consider using a soft link instead. A soft link (also known as **symbolic links** or **symlink**) is an entry that points to another directory or file. This is different to a hard link, because a hard link is a duplicate entry that references an inode, while a symbolic link references a specific path. Symbolic links can not only be moved around between filesystems; we can also create a symbolic link to a directory as well. To illustrate how a symbolic link works, let's create one. In my case, I'll delete `file3` and recreate it as a symbolic link. We'll again use the `ln` command:

```
rm file3
ln -s file1 file3
```

With the `-s` option of `ln`, I'm creating a symbolic link. First, I deleted the original hard link with the `rm` command (which doesn't disturb the original file, `file1`) and then created a symbolic link, also named `file3`. If we use `ls -i` again, we'll see that `file3` does not have the same inode number as `file1`:



```
jay@ubuntu-server:~$ rm file3
jay@ubuntu-server:~$ ln -s file1 file3
jay@ubuntu-server:~$ ls -i
529037 file1  530967 file2  530965 file3
jay@ubuntu-server:~$
```

Figure 5.10: Output of ls -i after creating a symbolic link

Notice that the inode numbers of each file are all different. At this point, the main difference compared with a hard link should become apparent. A symbolic link is not a clone of the original file; it's simply a pointer to the original file's path. Any commands you execute against `file3` are actually being run against the target that the link is pointing to. Hard links on the other hand, point directly to the file.

In practice, symbolic links are incredibly useful when it comes to server administration. However, it's important not to go crazy and create a great number of symbolic links all over the filesystem. This certainly won't be a problem for you if you are the only administrator on the server, but if you resign and someone takes your place, it will be a headache for them to figure out all of your symbolic links and map where they lead to. You can certainly create documentation for your symbolic links, but then you'd have to keep track of them and constantly update documentation. My recommendation is to only create symbolic links when there are no other options, or if doing so benefits your organization and streamlines your file layout.

Getting back to the subject of symbolic links versus hard links, you're probably wondering which one you should use and when to use it. The main benefit of a hard link is that you can move either file (the link or the original) to anywhere on the same filesystem and the link will not break. This is not true of symbolic links; however, if you move the original file, the symbolic link will be pointing to a file that no longer exists at that location. Hard links are basically duplicate entries pointing to the same object, and thus have the same inode number, so both will have the same file size and content. A symbolic link is a pointer to the file's path—nothing more, nothing less.

However, even though I just spoke about the several benefits of hard links, I actually recommend symbolic links for most use cases. They can cross filesystems, can be linked to directories, and it's easier to determine from the output where they lead. If you move hard links around, you may forget where they were originally located or which file actually points to which. Sure, with a few commands you can find them and map them easily. But overall, symbolic links are more convenient in the long run. As long as you're mindful of recreating your symbolic link whenever you move the original file (and you use them only when you need to), you shouldn't have an issue.

## Summary

In this chapter, we expanded our terminal kung-fu to another level and looked at concepts such as moving and copying files. We continued into a discussion of two popular text editors, Nano and Vim. Then, we took a dive into the subject of streams, and finished off the chapter with an understanding of the differences between symbolic and hard links, as well as how to create them.

In *Chapter 6*, *Boosting Your Command-Line Efficiency*, we'll dive a bit deeper into command-line tips and tricks, which will include a discussion on Bash history, writing basic scripts, and more.

## Further reading

- Vim Text Editor Tutorial Series: `https://learnlinux.link/vim`

# 6

# Boosting Your Command-line Efficiency

Throughout this book so far, we've been using the command line quite heavily. Using the shell, we've installed packages, created users, edited configuration files, and more. In the last chapter, we took a look at file management to enhance our terminal skills further. This time around, we dedicate an entire chapter to the shell, with the goal of becoming more efficient with it. Here, we'll take what we already know and add some useful time-saving tips, some information on looping, variables, and we'll even look into writing scripts.

In this chapter, we will cover the following topics:

- Understanding the Linux shell
- Understanding Bash history
- Learning some useful command-line tricks
- Understanding variables
- Writing simple scripts
- Putting it all together: Writing an `rsync` backup script

Let's begin this chapter with further discussion regarding the Linux shell, which will help us better understand how we're interacting with the server while we enter commands.

## Understanding the Linux shell

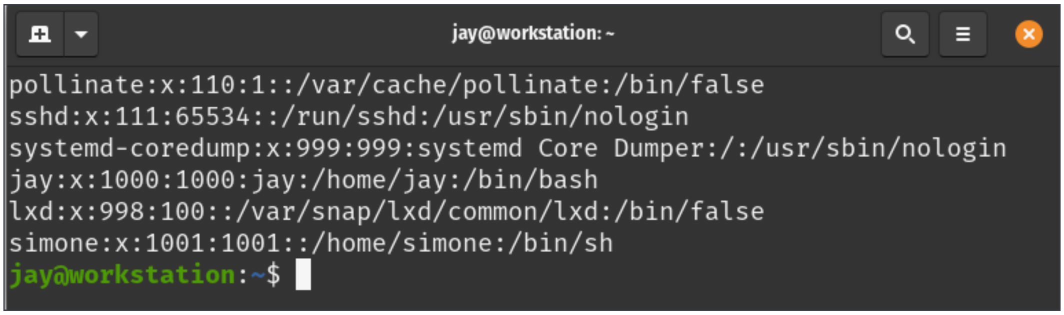
When it comes to the Linux shell, it's important to understand what exactly the term pertains to. We've been using the command line repeatedly throughout the book, but we haven't yet had any formal discussion about the actual interface through which our commands are entered.

Essentially, we've been entering our commands into a command interpreter known as the **Bourne Again Shell**, or simply **Bash**. Bash is just one of many different *shells* that you can use to enter commands. There are other options, including **Zsh**, **Fish**, and **ksh**, but Bash is the default command shell for the majority of Linux distributions. It's even available on macOS (although the default on that platform is Zsh nowadays), as well as in Windows by installing the Windows Subsystem for Linux. Therefore, by understanding the basics of Bash, your knowledge will be compatible with other distributions and platforms. While it's fun to learn other shells such as Zsh, Bash is definitely the one to focus the most attention on if you're just starting.

You may wonder, then, where you configure the shell that your user account will use. If you recall from *Chapter 2*, *Managing Users and Permissions*, we looked at the `/etc/passwd` file. As I'm sure you remember, this file keeps a list of user accounts available on the system. Go ahead and take a look at this file to refresh yourself by entering the following command:

```
cat /etc/passwd
```

This will produce an output like the one shown in *Figure 6.1*:



```
pollinate:x:110:1::/var/cache/pollinate:/bin/false
sshd:x:111:65534::/run/sshd:/usr/sbin/nologin
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
jay:x:1000:1000:jay:/home/jay:/bin/bash
lxd:x:998:100::/var/snap/lxd/common/lxd:/bin/false
simone:x:1001:1001::/home/simone:/bin/sh
jay@workstation:~$
```

Figure 6.1: The last several lines of a sample /etc/passwd file

See the last field in every entry? That's where we configure which shell is launched when a user logs in or starts a new terminal session. Unless you've already changed it, the entry for your user account should read `/bin/bash`. You'll see other variations in this file, such as `/bin/false` or `/usr/sbin/nologin`.

Setting the user's shell to one of these will actually prevent them from being able to log in. This is primarily used by system accounts, those that exist to run a task on the system, but we don't want to allow such accounts to be used by users for security reasons (the less an account can do, the safer). The `/usr/sbin/nologin` shell also doesn't allow the user to log in, but will provide a polite message letting them know.

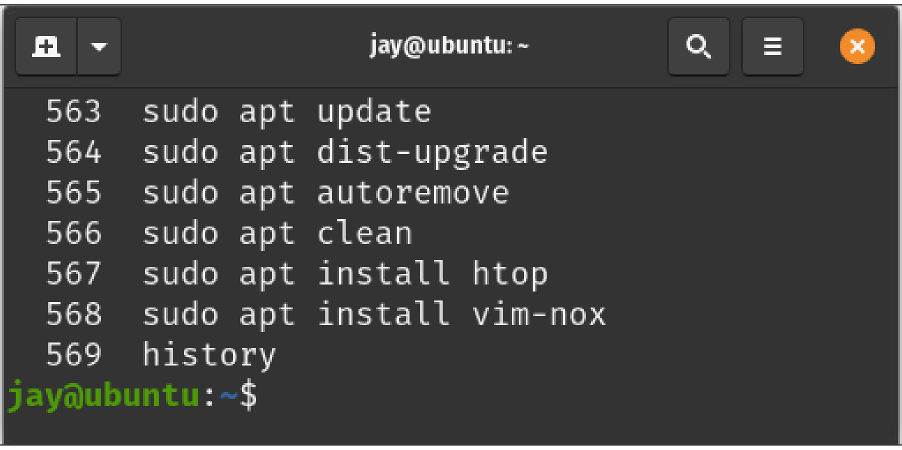
The shell program itself is responsible for reading the commands you type and having the Linux kernel execute them. Some shells, Bash notably, have additional features, such as *history*, that are very useful to administrators.

## Understanding Bash history

Speaking of history, let's dive right into that concept. By default, Bash keeps track of all the commands you enter during your sessions, so that if you need to recall a previously entered command, you can definitely do so. History also serves another purpose, and that is seeing what other users have been up to. However, since users can edit their own history to cover their tracks, it's not always useful for that purpose.

You may have already seen Bash's history feature in some form, if you've ever pressed the up and down arrows on the shell to recall a previously used command. If you didn't already know you can do that, go ahead and give it a try. You should see that by pressing the up and down arrows, you can cycle through commands that you've used previously.

Another trick is that you can also simply type `history` in the shell and see a list of previously entered commands, as shown in *Figure 6.2*:



```
  563  sudo apt update
  564  sudo apt dist-upgrade
  565  sudo apt autoremove
  566  sudo apt clean
  567  sudo apt install htop
  568  sudo apt install vim-nox
  569  history
jay@ubuntu:~$
```

Figure 6.2: Output from the history command

At this point, you can copy and paste a command you've used previously from this list to run it again. In fact, there's an even easier way. Do you notice the number on the left of each command? We can utilize that number to quickly recall a previously used command. In my screenshot, item `563` is where I ran `sudo apt update`. If I wanted to run that same command again, I can simply enter the following command:

```
!563
```

In this case, I typed just four characters, and I was able to recall the previously used command, which performs the same action as typing this:

```
sudo apt update
```

That saves a lot of typing, which is great because we administrators want to type as little as possible (unless we're writing a book).

Let's look at a few additional history commands we can use. First, if we want to delete something from the history, we can simply do this:

```
history -d 563
```

In this example, we deleted item `563` from Bash's history. To delete a different history entry, simple replace `563` with whatever the number is for the item we would want to remove. You may be wondering why deleting something from the history would be necessary. The answer to that is simple: sometimes we make mistakes. Perhaps we mistyped something, and we don't want a junior administrator to look at the history and rerun an invalid command. Worse, if we accidentally saved a password to the history, it will be there for all to see. We would definitely want to remove that item so that the password isn't saved in plain text in the history file. One very common example of this is with MySQL or MariaDB. When you enter the MySQL or MariaDB shell, you can use the `-p` option and type the password in one line. It would look something like this:

```
mariadb -u root -pSuperSecretPassword
```

That command may appear useful, because in one command you'd be logged in to your database server as `root`. However, this is one of my pet peeves—I really don't like it when people run commands that include a password in the clear. Having the `root` password in your shell's history is a HUGE security risk. This is just one example of something you won't want in our Bash history, though. My main goal here is to demonstrate that you should think about security when entering commands. If you have a potentially sensitive item in your command history, you should remove it. In fact, you can actually enter a command and not have it saved in the history at all. Simply prefix the command with a space. If you do, it will not be recorded in the history file. Go ahead, give it a try, and see for yourself.

Having commands prefixed with a space ignored in Bash is actually a custom option enabled by default in Ubuntu Server. Not all distributions include this feature. If you're using a distribution that doesn't have this feature, add the following to your `.bashrc` file (we will talk about this file in greater detail later).

```
HISTCONTROL=ignoreboth
```

This configuration line also causes duplicate commands to not be entered into the history file as well, which can condense the history file.

So, you might be wondering, where is this history information actually stored? Check out the `.bash_history` file, which is found in your home directory (or `/root` for the `root` user). When you exit your shell, your history is copied to that file. If you remove that file, you're effectively clearing your history. I don't recommend you make a habit of that, though. Having a history of commands is very useful, especially when you may not remember how you solved a problem last time. History in Bash can save you from looking up a command again. To find out more about what the `history` command can do, check out its man page with `man history`.

Learning new tricks with the command line that allow me to work more efficiently is a great feeling, at least for me. In the next section, we'll explore some useful tricks we can utilize while working with the shell.

## Learning some useful command-line tricks

Productivity hacks utilizing the shell are some of my favorite things in this world, right up there with music, video games, and Diet Pepsi. There's nothing like the feeling you get when you discover a useful feature that saves you time or increases your efficiency. There are many things I've discovered along the way that I wish I had known earlier on. One of my goals while writing this book is to teach you as many things as I can that took me longer to learn than I'm comfortable admitting to. In this section, in no particular order, I'll go over a few tricks that increased my workflow.

First, entering `!!` (two exclamation marks) in your terminal will repeat the command you last used. By itself, this may not seem like much. After all, you can press the up arrow key once and press *Enter* to recall the previous command and execute it. But, when paired with `sudo`, `!!` becomes more interesting. Imagine for a moment that you entered a command that needs `root` privileges, but you forgot to use `sudo`. We've all made this mistake. In fact, as of the time I'm writing this chapter, I've been using Linux for 18 years and I *still* forget to use `sudo` from time to time. When we forget `sudo`, we have to type the command all over again. Or, we can just do this:

```
sudo !!
```

And just like that, you prefixed the previously used command with `sudo` without having to completely retype it.

Speaking of avoiding unnecessary typing, a very easy (yet incredibly useful) feature is **Tab Completion**. Often, the Bash shell will be able to automatically complete part of your commands. If you start typing a few characters of a command or path, press *Tab* on your keyboard, and if the characters you've typed are enough to narrow down the result, the shell will complete the path for you. You can also press *Tab* twice in succession to see a list of possibilities that match the characters you've typed so far. Go ahead and give it a shot. By way of a quick example, you can type `ls` along with the path to your home directory, leaving some characters out on purpose. Then press *Tab*, and see whether the command automatically completes. For example, I can type the following into the terminal:

```
ls /home/j
```

And after I press *Tab*, it completes the command for me:

```
ls /home/jay
```

In addition, there are other special keyboard keys that will help you to navigate the command line quicker. Here's a table containing some of the most useful keyboard shortcuts:

| **Keyboard shortcut** | **Result** |
|---|---|
| *Ctrl* + *a* | Moves the cursor to the beginning of the line |
| *Ctrl* + *e* | Moves the cursor to the end of the line |
| *Ctrl* + *l* | Clears the screen |
| *Ctrl* + *k* | Deletes characters from the cursor to the end of the line |
| *Ctrl* + *u* | Deletes everything you've typed on that line (also works to clear text while entering a password) |

Going a bit further into the command history, we can also press *Ctrl* + *r* on the shell to initiate a search. After pressing these keys, we can start typing a command, and we'll get a preview of a command that matches what we're typing, which will be narrowed down further as we type more characters of it. This is one of those things that is hard for me to describe, and screenshots certainly don't help here, so go ahead and just give it a shot. For example, press *Ctrl* + *r* and then start typing `sudo apt`. The last time you used that command should appear, and you can press *Ctrl* + *r* again, and again, and again to see additional examples of commands that you've typed in the past that contain those characters. When you get efficient with this, it's actually faster than the `history` command, but it takes a bit to get used to this.

Another fun trick is editing a command you've previously typed in a text editor. I know this sounds strange, but hear me out. Assume you pressed the up arrow, you have a very long command, and you just want to edit part of it without having to execute the entire thing, perhaps a command like this:

```
sudo apt update && sudo apt install apache2
```

Let's assume you want to install `nginx` instead of `apache2`, but the rest of the command is right. If you hold *Ctrl* and then press *x* followed by *e*, the command will open in a text editor. There, you can change the command. Once you're done making your changes, the command will execute once you save the file. Admittedly, this is usually only useful when you have a very long command and you need to just change part of it. It's also a little weird, but so are computers.

Did you notice the two `&` symbols in the previous command? This is another useful trick; you can actually chain commands together. In the previous example command, we're telling the shell to execute `sudo apt update`. Next, we're telling the shell to then execute `sudo apt install apache2`. The double ampersand is known as the logical `AND` operator, so the second command is run if the first was successful. If the first command was successful, the second command will execute right after. Another method to chain commands is this:

```
sudo apt update; sudo apt install apache2
```

The difference with the semicolon is that we're telling the shell to execute the second command *regardless* of whether the first command was successful. You may then be wondering, what constitutes success on the shell? An obvious answer to this question might be "it's successful if there are no error messages." While that's true, the shell utilizes exit codes to programmatically attribute success or failure. You can see the exit code of a command by typing this immediately after the previous command finishes:

```
echo $?
```

An exit code of `0` means success; anything else is some sort of error. Different programs will attribute different codes to different types of failures, but `0` is always a success. With this command, what we're actually doing is printing the content of a variable. `$?` is actually a variable, which in this case only exists to hold an exit code. The `echo` command itself can be used to print text to the shell, but it's often used to print the contents of a variable (we'll get into this in more detail in the *Understanding variables* section).

Now, it's time for my favorite time-saving trick of them all—command aliases. The concept of an alias is simple: it allows you to create a command that is just another name for another command. This allows you to simplify commands down to just one word or a few letters. Consider this command, for example:

```
alias install="sudo apt install"
```

When you enter a previous command, you will receive no actual output. But what happens is now you have a new command available—`install`. This command isn't normally available; you just created it with this command.

> You can verify that the alias was created successfully by simply running the `alias` command, which will show you a list of aliases present in the shell. If you create a new alias, you should see it in the output. You'll also see additional aliases in the output that you did not create. This is because Ubuntu sets up some by default. In fact, even the `ls` command is an alias!

With this new alias created, any time you execute `install` on the command line, you're instead executing `sudo apt install`. Now, installing packages becomes simpler:

```
install tmux
```

Just like that, you installed `tmux`. You didn't have to type `sudo apt install tmux`; you just simplified the first three words in the command into `install`. In fact, you can simplify it even further:

```
alias i='sudo apt install'
```

Now, you can install a package with this:

```
i tmux
```

With aliases, you can get very creative. Here are some of my personal favorites:

View the top 10 CPU consuming processes:

```
alias cpu10='ps -L aux | sort -nr -k 3 | head -10'
```

View the top 10 RAM consuming processes:

```
alias mem10='ps -L aux | sort -nr -k 4 | head -10'
```

View all mounted filesystems, and present the information in a clean tabbed layout:

```
alias lsmount='mount |column -t'
```

Clear the screen by simply typing `c`:

```
alias c=clear
```

What other aliases can you come up with? Think of a command you may use on a regular basis and simplify it.

There's one issue though, and that is the fact that when you exit your terminal window, your aliases are wiped out. How do you retain them? That leads me into my next productivity trick, editing your `.bashrc` file. This file is present in your home directory and is read every time you start a new terminal session. You can add all of your `alias` commands there; just add them somewhere in the file (for example, at the end). You will need to include the entire command, beginning with `alias` and ending with the commands in quotes. If you wanted to steal my example aliases, you would enter the following lines somewhere in your `.bashrc` file:

```
alias i='sudo apt install'
alias cpu10='ps -L aux | sort -nr -k 3 | head -10'
alias mem10='ps -L aux | sort -nr -k 4 | head -10'
alias lsmount='mount |column -t'
```

There are, of course, additional time-saving tricks that we could talk about here, but then again, Bash is so complex that we can write an entire book about it (and many people have). As we go along in this chapter, I'll give you even more tips. For now, here's a final trick, which changes your working directory back to the previous directory you were in:

```
cd -
```

That simple command was mentioned in *Chapter 4*, *Navigating and Essential Commands*, but it's worth a second mention—you're welcome! Next, let's take a look at shell variables, which allow us to store information for easy access in other commands.

## Understanding variables

Bash is more than just a shell. You could argue that it is very similar to a complete programming language, and you wouldn't be wrong. Bash is a scripting engine (we will get into scripting later) and there are many debates as to what distinguishes a scripting language from a programming language, and that line becomes more and more blurred as new languages come out. As with any scripting language, Bash supports variables. The concept of variables is very simple in Bash, but I figured I'd give it its own (relatively short) section to make sure you understand the basics. You can set a variable with a command such as the following:

```
myvar='Hello world!'
```

When Bash encounters an equal sign after a string, it assumes you're creating a variable. Here, we're creating a variable named `myvar` and setting it equal to `Hello world!` Whenever we refer to a variable, though, we need to specifically clarify to Bash that we're requesting a variable, and we do that by prefixing it with a dollar symbol (`$`). Consider this command:

```
echo $myvar
```

If you've set the variable as I have, executing that command will print `Hello world!` to `stdout`. The `echo` command is very useful for printing the contents of variables. The key thing to remember here is that when you *set* a variable, you don't include the `$` symbol, but you do when you retrieve it. Also, keep in mind that there's no space on either side of the equals sign.

You will see variations of variable name formats as you work with various Linux servers. For example, you may see variable names in all caps, camel case (`MyVar`), as well as other variations. These variations are all valid, and depending on the background of the individual creating them (developers, administrators, and so on), you may see different forms of variable naming.

Variables work in other aspects of the shell as well, not just with `echo`. Consider this:

```
mydir="/etc"
ls $mydir
```

Here, we're storing a directory name in a variable, and using the `ls` command against it to list the contents of it. This may seem relatively useless, but when you're scripting, this will save you time. Anytime you need to refer to something more than once, it should be in a variable. That way, in a script, you can change the contents of that variable just one time and everywhere in the script will reference it.

There are also variables that are automatically present in your shell that you did not explicitly set yourself. Enter this command for fun:

```
env
```

Wow! You should see a lot of variables, especially if you enter it in a desktop version of Ubuntu.

These variables are set by the system, but can still be accessed via `echo` as you would any other. Some notable ones include `$SHELL` (stores the name of the binary that currently handles your shell), `$USER` (stores your current username), and `$HOST` (stores the hostname for your device). Any of these variables can be accessed at any time, and may even prove beneficial in scripts.

We've already gone over **standard output** (**stdout**), **standard error** (**stderr**), and **standard input** (**stdin**) in the previous chapter. We'll use standard input again here, when we capture input to store it as a variable. Try this command for example:

```
read age
```

When you run this command, you'll just be brought to a blank line, with no indication as to what you should be doing. Go ahead and enter your age, and then press *Enter*. Next, run this:

```
echo $age
```

In a script, you would want to inform the user what they should be entering, so you would probably use something similar to these commands:

```
echo "Please enter your age"
read age
echo "Your age is $age"
```

We've discussed standard input in the previous chapter, and we can see it in action again here as we capture input from the user and store it in a variable.

Automation is a subject we'll explore multiple times throughout the remainder of the book, which will include more advanced subjects such as configuration management. Writing scripts is the simplest form of automation, which gives you the ability to type commands in a text file and have them all execute. That's what we'll explore next.

## Writing simple scripts

This is the section where everything we've talked about so far starts to come together. Scripting can be very fun and rewarding, as they allow you to automate large jobs or just simplify something that you find yourself doing over and over. The most important point about scripting is this: if it's something you'll be doing more than once, you really should be making it into a script. This is a great habit to get into.

A script is a very simple concept; it's a just a text file that contains commands for your shell to execute one by one. A script written to be executed by Bash is known as a Bash script, and that's what we'll work on creating in this section.

At this point, I'm assuming that you've practiced a bit with a text editor in Linux. It doesn't matter if you use Vim or Nano. Since we've edited text files before (we went over that in *Chapter 5*, *Managing Files and Directories*), I'm under the assumption that you already know how to create and edit files. We'll be using a text editor to create a simple script as an example, using the following command:

```
nano ~/myscript.sh
```

If you weren't already aware, a tilde (~) is just a shortcut for a user's home directory. Therefore, on my system, the previous command would be the same as if I had typed:

```
nano /home/jay/myscript.sh
```

Inside the file, type the following:

```
#!/bin/bash

echo "My name is $USER"
echo "My home directory is $HOME"
echo "My default text editor is $EDITOR"
```

Save the file and exit the editor. In order to run this file as a script, we need to mark it as executable:

```
chmod +x ~/myscript.sh
```

To execute it, we simply call the path to the file and the filename:

```
~/myscript.sh
```

The output should look similar to the following:

```
My name is jay
My home directory is /home/jay
My default text editor is vim
```

The first line, `#!/bin/bash`, might seem strange if you haven't seen it before. Normally, lines starting with a hash symbol (`#`) are ignored by the interpreter. The one on the first line is an exception to this. The `#!/bin/bash` entry we see on the first line is known as a **hash bang**, or **shebang**. Basically, it just tells the kernel which interpreter to use in order to run the commands inside the script. There are other interpreters we could be using, such as `#!/usr/bin/python` if we were writing a script in the Python language. Since we're writing a Bash script, we used `#!/bin/bash`.

The lines that followed were simple print statements. Each one used a system variable, so you didn't have to declare any of those variables as they already existed. Here, we printed the current user's username, home directory, and default text editor.

The concept of scripting becomes more valuable when you start to think of things you do on a regular basis that you can instead automate. To be an effective Linux administrator, it's important to adopt the automation mindset. Again, if you are going to do a job more than once, script it. Here's another example script to help drive this concept home. This time, the script will actually be somewhat useful:

```
#!/bin/bash

sudo apt install -y apache2
sudo apt install -y libapache2-mod-php7.4
sudo a2enmod php7.4
sudo systemctl restart apache2
```

What we've done here is theoretically scripted the setup of a web server. We could extend this script further by having it copy site content to `/var/www/html`, enable a configuration file, and so on. But from the preceding script, you can probably see how scripting can be useful in condensing the amount of work you do. This script could be an advanced web server install script that you could simply copy to a new server and then run.

Notice that the example uses the `-y` option with `apt`. If you weren't already aware, that automatically answers *yes* to prompts that may come up as part of the process. Scripts are typically not interactive, meaning there may not be an administrator sitting in front of it to answer prompts when they appear. Also, using the `a2enmod` command to enable `php7.4` was not really necessary, as it would've been enabled automatically as part of installing the `libapache2-mod-php7.4` package. But I think you get the idea; we want to be explicit in scripts and type instructions for the exact state we want things to be in.

Now, let's get a bit more advanced with scripting. The previous script only installed some packages, something we probably could've done just as easily by copying and pasting the commands into the shell. Let's take this script a bit further. Let's write a conditional statement. Here's a modified version of the previous script:

```
#!/bin/bash

# Install Apache if it's not already present
if [ ! -f /usr/sbin/apache2 ]; then
    sudo apt install -y apache2
    sudo apt install -y libapache2-mod-php7.4
    sudo a2enmod php7.4
    sudo systemctl restart apache2
fi
```

Now it's getting a bit more interesting. The first line after the hash bang is a comment, letting us know what the script does:

```
# Install Apache if it's not already present
```

Comments are ignored by the interpreter, but are useful in letting us know what a block of code is doing.

Next, we start an `if` statement:

```
if [ ! -f /usr/sbin/apache2 ]; then
```

Bash, like any scripting language, supports branching and the **If Statement** is one way of doing that. Here, it's checking for the existence of the `apache2` binary. The `-f` option here specifies that we're looking for a file. We can change this to `-d` to check for the existence of a directory instead. The exclamation mark is an inverse. It basically means we're checking if something is *not* present. If we wanted to check if something *is* present, we would omit the exclamation mark. Basically, we're setting up the script to do nothing if Apache is already installed. In this case, inside the brackets we are just executing a shell command, and then the result is checked. The commands sandwiched inside the `if` statement are simply installing packages.

Finally, we close out our `if` statement with the word *if* backward (`fi`). If you forgot to do this, the script will fail.

With regard to the concept of `if` statements, we can compare values as well. Consider the following example:

```
#!/bin/bash

myvar=1

if [ $myvar -eq 1]; then
    echo "The variable equals 1"
fi
```

With this script, we're merely checking the contents of a variable, and taking action if it equals a certain number. Notice we didn't use quotation marks when creating the variable, since we just set a number (integer) here. We would've only used quotation marks if we wanted to set the variable value to a string. We can also take action if the `if` statement doesn't match:

```
#!/bin/bash

myvar=10

if [ $myvar -eq 1]; then
    echo "The variable equals 1"
else
    echo "The variable doesn't equal 1"
fi
```

This was a silly example, I know, but it works as far as illustrating how to create an `if/else` logic block in Bash. The `if` statement checks to see whether the variable was equal to `1`. It isn't, so the `else` block executes instead.

The `-eq` portion of the command is similar to `==` in most programming languages. It's checking to see whether the value is equal to something. Alternatively, we can use `-ne` (not equal), `-gt` (greater than), `-ge` (greater than or equal to), `-lt` (less than), and so on.

At this point, I recommend you take a break from reading to further practice scripting (practice is key to committing concepts to memory). Try the following challenges:

- Ask the user to enter input, such as their age, and save it to a variable. If the user enters a number less than 30, tell them they're young. If the number is equal to or greater than 30, `echo` a statement telling them that they're old.
- Write a script that copies a file from one place to another. Make the script check to see whether that file exists first, and have an `else` statement printing an error if the file doesn't exist.
- Think about any topic we've already worked on during this book, and attempt to automate it.

Now, let's take a look at another concept, which is looping. The basic idea behind looping is simply doing something repeatedly until a condition has been met. Consider the following example script:

```
#!/bin/bash

myvar=1

while [ $myvar -le 15 ]
do
    echo $myvar
    ((myvar++))
done
```

Let's go through the script line by line to understand what it's doing.

```
myvar=1
```

With this new script, we're creating a control variable, called `myvar`, and setting it equal to `1`:

```
while [ $myvar -le 15 ]
```

Next, we set up a `while` loop. A `while` loop will continue until a condition is met. Here, we're telling it to execute the statements in the block over and over until `$myvar` becomes equal to `15`. In fact, a `while` loop can continue forever if you enter something incorrectly, which is known as an **Infinite Loop**. An infinite loop is dangerous, and can cause your server to stop responding. If you used `-ge 0` instead, you would've created exactly that:

```
do
```

With `do`, we're telling the `for` loop to prepare itself to start doing something:

```
    echo $myvar
```

Here, we're printing the current content of the `$myvar` variable—nothing surprising here:

```
    ((myvar++))
```

With this statement, we're using what's known as an incrementor to increase the value of our variable by `1`. The double parenthesis tells the shell that we're doing an arithmetic operation, so the interpreter doesn't think that we're working with strings:

```
done
```

When we're done writing a `while` loop, we must close the block with `done`. If you've typed the script properly, it should count from `1` to `15`.

Another type of loop is a **For Loop**. A `for` loop executes a statement for every item in a set. For example, you can have the `for` loop execute a command against every file in a directory. Consider this example:

```
#!/bin/bash

turtles='Donatello Leonardo Michelangelo Raphael'
for t in $turtles
do
    echo $t
done
```

Let's take a deeper look into what we've done here:

```
turtles='Donatello Leonardo Michelangelo Raphael'
```

Here, we're creating a list and populating it with names. Each name is one item in the list. We're calling this list `turtles`. We can see the contents of this list with `echo` as we would with any other variable:

```
echo $turtles
```

Next, let's look at how we set up the `for` loop:

```
for t in $turtles
```

Now, we're telling the interpreter to prepare to do something for every item in the list. The `t` here is arbitrary, we could've used any letter here or even a longer string. We're just setting up a temporary variable we want to use in order to hold the current item the script is working on.

```
do
```

With `do`, we're telling the `for` loop to prepare itself to start doing something:

```
    echo $t
```

Now, we're printing the current value of `$t` to `stdout`:

```
done
```

Just as we did with the `while` loop, we type `done` to let the interpreter know this is the end of the `for` loop. Effectively, we just created a `for` loop to print each item in a list independently:

```
Donatello
Leonardo
Michelangelo
Raphael
```

We included four turtle names in our list, and we were able to iterate through them and print them out, one by one.

As much as I love turtles (especially the teenage mutant ninja variety), that script isn't very practical or useful to us for server administration. Next, we're going to write a script that can actually be quite useful.

## Putting it all together – Writing an rsync backup script

Let's close this chapter with a Bash script that will not only prove to be very useful, but will also help you enhance your skills. The `rsync` utility is one of my favorites; it's very useful for not only copying data from one place to another, but also helpful for setting up a backup job. Let's use the following example `rsync` command to practice automation:

```
rsync -avb --delete --backup-dir=/backup/incremental/08-17-2020 /src /
target
```

This example `rsync` command uses the `-a` (archive) option, which retains the metadata of the file(s) it copies to the target, such as the timestamp and owner. The `-v` option gives us verbose output, so we can see exactly what `rsync` is doing. The `-b` option enables backup mode, which means that if a file on the target will be overwritten by a file from the source, the previous version of that file will be renamed so it won't be overwritten. Combining these three options, we simplify it into `-avb` rather than typing `-a -v -b`. The `--delete` option tells `rsync` to delete any files in the target that aren't present in the source (since we used `-b`, any file that is deleted will be retained). The `--backup-dir` option tells `rsync` that any time a file would have been renamed in this way (or deleted), to instead just copy it to another directory. In this case, we send any files that would have been overwritten to the `/backup/incremental/08-16-2020` directory.

Let's script this `rsync` job. One problem we can fix in our script right away is the date that is present inside the directory we're using for the `--backup-dir`. The date changes every day, so we shouldn't be hardcoding this. Therefore, let's start our script by addressing this:

```
#/bin/bash

CURDATE=$(date +%m-%d-%Y)
```

We're creating a variable called `CURDATE`. We're setting it equal to the output of the `$(date +%m-%d-%Y)` command. You can execute `date +%m-%d-%Y` in your terminal window to see exactly what that does. In this case, putting a command (such as `date`) in parentheses and a dollar symbol means that we're executing the command in a **sub-shell**. The command will run, and we're going to capture the result of that command and store it in the `CURDATE` variable.

Next, let's make sure `rsync` is actually installed, and install it if it's not:

```
If [ ! -f /usr/bin/rsync ]; then
    sudo apt install -y rsync
fi
```

Here, we're simply checking to see whether `rsync` is *not* installed. If it's not, we'll install it via `apt`. This is similar to how we checked for the existence of `apache2` earlier in this chapter.

Now, we add the final line:

```
rsync -avb --delete --backup-dir=/backup/incremental/$CURDATE /src /
target
```

You can definitely see the magic of variables in Bash now, if you haven't already. We're including `$CURDATE` in our command, which is set to whatever the current date actually is. When we add it all together, our script looks like this:

```
#/bin/bash

CURDATE=$(date +%m-%d-%Y)

if [ ! -f /usr/bin/rsync ]; then
    sudo apt install -y rsync
fi

rsync -avb --delete --backup-dir=/backup/incremental/$CURDATE /src /
target
```

This script, when run, will run an `rsync` job that will copy the contents from `/src` to `/target`. (Be sure to change these directories to match the source directory you want to back up and the target where you want to copy it to.) The beauty of this is that `/target` can be an external hard drive or network share. So, in a nutshell, you can automate a nightly backup. This backup, since we used the `-b` option along with `--backup-dir`, will allow you to retrieve previous versions of a file from the `/backup/incremental` directory. Feel free to get creative here as far as where to place previous file versions and where to send the backup.

Of course, don't forget to mark the script as executable, assuming it was saved with a name like `backup.sh`:

```
chmod +x backup.sh
```

At this point, you can put this script in a cron job to automate its run. To do so, it's best to put the script in a central location where it can be found, such as in `/usr/local/bin`:

```
mv backup.sh /usr/local/bin
```

You can consider creating a cron job for this script to be run periodically. We'll cover that in *Chapter 7*, *Controlling and Monitoring Processes*, which is the very next chapter. With a cron job, you can set up various tasks to run at different times in order to make your server essentially do your work for you.

## Summary

In this chapter, we dived into a number of more advanced concepts relating to shell commands, such as redirection, Bash history, command aliases, some command-line tricks, and more. Working with the shell is definitely something you'll continue to improve upon, so don't be worried if you have any trouble committing all of this knowledge to memory. After over 18 years working with Linux, I'm still learning new things myself. The main takeaway in this chapter is to serve as a starting point to broaden your command-line techniques, and also serve as the basis for future exploration into the subject.

In the next chapter, we'll take a look at managing processes, which will include job management, taming misbehaving processes, and more. See you there!

## Further reading

- Comparison operators for Bash: `http://tldp.org/LDP/abs/html/comparison-ops.html`
- Commandlinefu: `https://www.commandlinefu.com/commands/browse`
- Bash reference manual: `https://www.gnu.org/software/bash/manual/html_node/index.html#SEC_Contents`

# 7
# Controlling and Managing Processes

On a typical Linux server, there can be over a hundred processes running at any given time. The purposes of these processes range from system services, such as the **Network Time Protocol** (**NTP**) service, to processes that serve information to others, such as the Apache web server. As an administrator of Ubuntu servers, you will need to be able to manage these processes, as well as manage the resources available to them. In this chapter, we'll take a look at process management, including the `ps` command, managing job control commands, and more.

As we work through these concepts, we will cover the following topics:

- Managing jobs
- Understanding the `ps` command
- Changing the priority of processes
- Dealing with misbehaving processes
- Managing system processes
- Scheduling tasks with `cron`

To begin our exploration of managing processes, let's take a look first at managing jobs. Not only will this help us understand the concepts better, but it will also provide us with a better understanding of backgrounding and foregrounding.

## Managing jobs

Up until now, everything we have been doing on the shell has been right in front of us, from execution to completion. We've installed applications, run programs, and walked through various commands. Each time, we've had control of our shell taken from us, and we've only been able to start a new task when the previous one had finished. For example, if we were to install the `vim-nox` package with the `apt install` command, we would watch helplessly while `apt` takes care of fetching the package and installing it for us. While this is going on, our cursor goes away and our shell completes the task for us without allowing us to queue up another command. We can always open a new shell to the server and multi-task by having two windows open at once, each doing different tasks. But that's likely not going to be the most efficient method of multitasking when working with the command line.

Instead, we can actually background a process without waiting for it to complete while working on something else. Then, we can bring that process back to the front to return to working on it or to check whether or not it finished successfully. Think of this as a similar concept to a windowing desktop environment, or user interfaces on the Windows or macOS operating systems. We can work on an application, minimize it to get it out of the way, and then maximize it to continue working with it. Essentially, that's the same concept of backgrounding a process in a Linux shell. If you were curious how I had a Vim process running in the background earlier while discussing prioritizing processes, what I did was send it to the background.

So how exactly do you background and foreground a process? This concept can be somewhat difficult to explain. In my opinion, the easiest way to learn a new concept is to try it out, and the easiest example I can think of is by (yet again) using a text editor. I promise that this time, using a text editor as an example won't be lame. In fact, this example is extremely useful and may just become a part of your daily workflow. To do this exercise, you can use any command-line text editor you prefer, such as Vim or Nano. On Ubuntu Server, `nano` is usually installed by default, so you already have it if you want to go with that. If you prefer to use Vim, feel free to install the `vim-nox` package if you haven't already installed it:

```
sudo apt install vim-nox
```

You can actually install `vim` rather than `vim-nox`, but I always default to `vim-nox` since it features built-in support for scripting languages.

Again, feel free to use whichever text editor you feel comfortable with. In the following examples, I'll be using `nano`, but if you use `vim`, just replace `nano` with `vim` every time you see it.

Anyway, to see backgrounding in action, open up your text editor. Feel free to open a file or just start a blank session. (If in doubt, type `nano` and press *Enter*.) With the text editor open, we can background it at any time by pressing *Ctrl* + *z* on our keyboard.

If you are using `vim` instead of `nano`, you can only background `vim` when you are *not* in **insert mode**, since it captures *Ctrl* + *z* rather than passing it to the shell.

Did you see what happened? You were immediately taken away from your editor and returned to the shell so you can now get back to executing commands. You should have seen some output similar to the following:

```
[1]+ Stopped nano
```

Here, we see the `job` number of our process, its status, and then the name of the process. Even though the process of your text editor shows a status of `Stopped`, it's still running. You can confirm this with the following command:

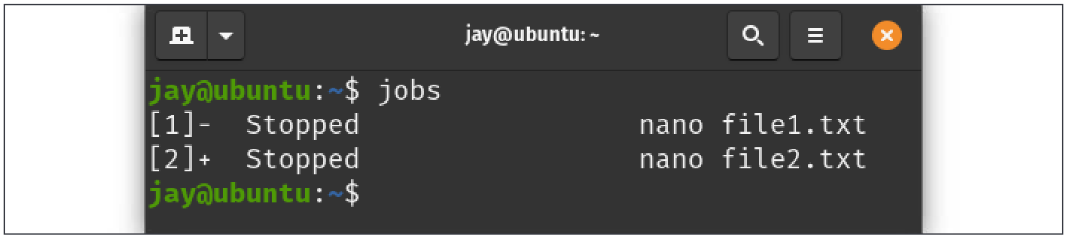
```
ps au |grep nano
```

In my case, I see the `nano` process running with a PID of `43231`:

```
jay        43231  0.0  0.1   5468  3632 pts/0    T    11:27   0:00 nano
```

At this point, I can execute additional commands, navigate around my filesystem, and get additional work done. When I want to bring my text editor back, I can use the `fg` command to foreground the process, which will resume it. If I have multiple background processes, the `fg` command will bring back the one I was working on most recently.

I gave you an example of the ps command to show that the process was still running in the background, but there's actually a dedicated command for that purpose, and that is the jobs command. If you execute the jobs command, you'll see in the output a list of all the processes running in the background:



```
jay@ubuntu:~$ jobs
[1]-  Stopped                 nano file1.txt
[2]+  Stopped                 nano file2.txt
jay@ubuntu:~$
```

Figure 7.1: Running the jobs command after backgrounding two nano processes

The output shows that I have two nano sessions in use, one modifying file1.txt, and the other modifying file2.txt. If I were to execute the fg command, that would bring up the nano session that's editing file2.txt, since that was the last one I was working in. That may or may not be the one I want to return to editing, though. Since I have the job ID on the left, I can bring up a specific background process by using its ID with the fg command:

```
fg 1
```

Knowing how to background a process can add quite a bit to your workflow. For example, let's say, hypothetically, that I'm editing a config file for a server application, such as Apache. While I'm editing this config file, I need to consult the documentation (man page) for Apache because I forgot the syntax for something. I could open a new shell and an SSH session to my server and view the documentation in another window. This could get very messy if I open up too many shells. It would be much simpler to background the current nano session, read the documentation, and then foreground the process with the fg command to return to working on it, all from one SSH session!

To background a process, you don't have to use *Ctrl* + *z*; you can actually background a process right when you execute it by entering a command with the ampersand symbol (&) typed at the end. To show you how this works, I'll use htop as an example. Admittedly, this may not necessarily be the most practical example, but it does work to show you how to start a process and have it backgrounded right away. We probably won't have htop installed yet, but for now feel free to install this package and then run it with the ampersand symbol:

```
sudo apt install htop
htop &
```

The first command, as you already know, installs the `htop` package on our server. With the second command, I'm opening `htop` but backgrounding it immediately. What I'll see when it's backgrounded is its job ID and process ID (more on this in the next section). Now, at any time, I can bring `htop` to the foreground with `fg`. Since I just backgrounded it, `fg` will bring `htop` back since it considers it the most recent. As you know, if it wasn't the most recent, I could reference its job ID with the `fg` command to bring it back even if it wasn't my most recently used job. Go ahead and practice using the ampersand symbol with a command and then bringing it back to the foreground. In the case of `htop`, it can be useful to start it, background it, and then bring it back anytime you need to check the performance of your server.

Keep in mind, though, that when you exit your shell, all your backgrounded processes will close. If you have unsaved work in your text editors, you'll lose what you were working on. For this reason, if you utilize background processes, you may want to check to see if you have any pending jobs still running by executing the `jobs` command before logging out.

In addition, you'll probably notice that some applications background cleanly, while others don't. In the case of using a text editor and `htop`, those applications stay paused in the background, allowing us to perform other tasks and then return to those commands later. However, some applications may still spit out diagnostic text regularly in your main window, whether they're backgrounded or not. To get even more control over your bash sessions, you can learn how to use a multiplexer, such as `tmux` or `screen`, to allow these processes to run in their own session such that they don't interrupt your work. Going over the use of a program such as `tmux` is beyond the scope of this book, but it is a useful utility to learn if you're interested.

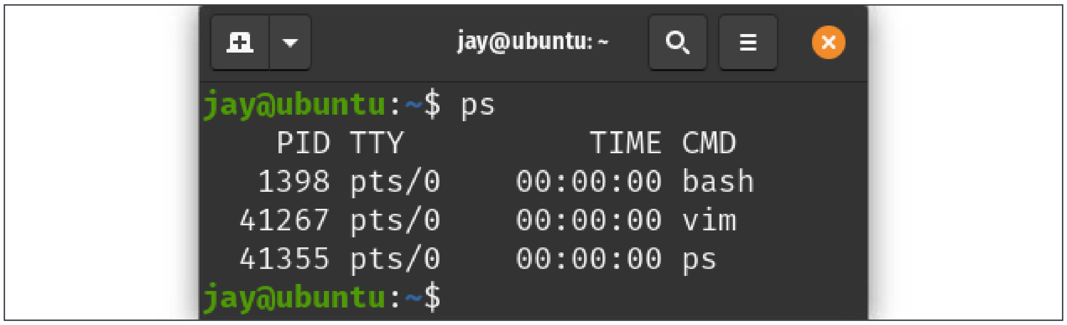
Being able to background and foreground a process allows us to manage tasks on the command line more effectively, and is definitely useful. Now, we can expand this and look at viewing other processes on the server, including those that we didn't manually start such as a text editor. In the next section, we'll take a look at the `ps` command, which can help us understand what is actually running on our server.

## Understanding the ps command

While managing our server, we'll need to understand what processes are running and how to manage them. Later in this chapter, we'll work through starting, stopping, and monitoring processes. But before we get to those concepts, we first need to be able to determine what is actually running on our server. The `ps` command allows us to do this.

## Viewing running processes with ps

When executed by itself, the `ps` command will show a list of processes run by the user who called the command:



```
jay@ubuntu:~$ ps
    PID TTY          TIME CMD
   1398 pts/0    00:00:00 bash
  41267 pts/0    00:00:00 vim
  41355 pts/0    00:00:00 ps
jay@ubuntu:~$
```

Figure 7.2: The output of the ps command, when run as a normal user and with no options

In *Figure 7.2*, you can see that when I ran the `ps` command as my own user with no options, it showed me a list of processes that I am running as myself. In this case, I have a `vim` session open (running in the background), and in the last line, we also see `ps` itself, which is also included in the output. We haven't gone through background processes yet, so don't worry about how I backgrounded `vim` just yet.

On the left side of the output, you'll see a number for each of the running processes. This is known as the **Process ID** (**PID**), which we mentioned in the *Managing jobs* section. Before we continue on, the PID is something that you really should be familiar with, so we may as well cover it right now.

Each process running on your server is assigned a PID, which differentiates it from other processes on your system. You may understand a process as `vim`, or `top`, or some other name. However, our server knows processes by their ID. When you open a program or start a process, it's given a PID by the kernel. As you work on managing your server, you'll find that the PID is useful to know, especially for the commands we'll be covering in this very chapter. If you want to kill a misbehaving process, for example, a typical workflow would be for you to find the PID of that process and then reference that PID when you go to kill the process (which I'll show you how to do in a later section). PIDs are actually more complex than just a number assigned to running processes, but for the purposes of this chapter, that's the main purpose we'll need to remember.

You can also use the `pidof` command to find the PID of a process if you know the name of it. For example, I showed you a screenshot of a `vim` process running with a PID of `41267`. You can also do so by running the following command:

```
pidof vim
```

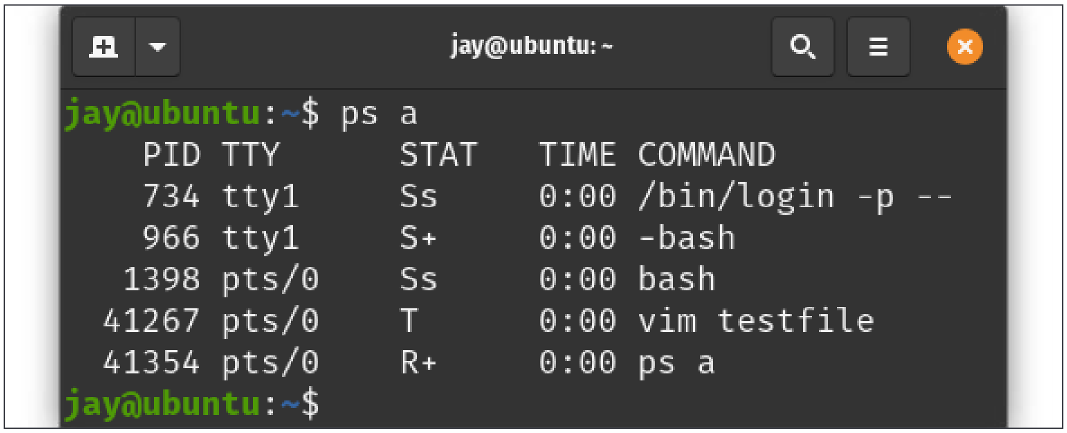
The output will give you the PID(s) of the process without you having to use the `ps` command.

## Configuring arguments to ps

Continuing with the `ps` command, there are several useful arguments you can give in order to change the way in which it produces an output. If you use the `a` option, you'll see more information than you normally would:

```
ps a
```

This will produce an output something like the following:



```
jay@ubuntu:~$ ps a
    PID TTY      STAT   TIME COMMAND
    734 tty1     Ss     0:00 /bin/login -p --
    966 tty1     S+     0:00 -bash
   1398 pts/0    Ss     0:00 bash
  41267 pts/0    T      0:00 vim testfile
  41354 pts/0    R+     0:00 ps a
jay@ubuntu:~$
```

Figure 7.3: The output of the ps a command

With `ps a`, we're seeing the same output as before, but with additional information, as well as column headings at the top. We now see a heading for `PID`, `TTY`, `STAT`, `TIME`, and `COMMAND`. From this new output, you can see that the `vim` processes I have running are editing a file named `testfile`. This is great to know, because if I had more than one `vim` session open and one of them was misbehaving, I would probably want to know which one I specifically needed to stop.

We already saw the `PID` and `COMMAND` fields, although we didn't see a formal heading at the top. The `PID` column we've already covered, so I won't go into any additional detail about that. The `COMMAND` field tells us the actual command being run, which is very useful if we either want to ensure we're managing the correct process or to see what a particular user is running (I'll demonstrate how to display processes for other users soon).

The `TTY`, `STAT`, and `TIME` fields are new; we didn't see those when we ran `ps` by itself. The `STAT` field gives us the status code of the process, which refers to which state the process is currently in. The state can be uninterruptible sleep (`D`), defunct (`Z`), stopped (`T`), interruptible sleep (`S`), and in the run queue (`R`). There is also paging (`W`), but that is not used anymore, so there's no need to cover it. Uninterruptible sleep is a state in which a process is generally waiting on input and cannot handle additional signals (we'll briefly talk about signals later on in this chapter). A defunct process (also referred to as a zombie process) has, to all intents and purposes, finished its job but is waiting on the parent to perform cleanup. Defunct processes aren't actually running, but remain in the process list and should normally close on their own. If such a process remains in the list indefinitely and doesn't close, it can be a candidate for the `kill` command, which we will discuss later. A stopped process is generally a process that has been sent to the background, which will be discussed in the next section. Interruptible sleep means that the program is idle: it's waiting for input in order to awaken.

The `TTY` column tells us which TTY the process is attached to. A TTY refers to a **teletypewriter**, which is a term used from a much different time period. In the past, a teletypewriter would be used to electronically send signals to a typing device on the other end of a wire. Obviously, we don't use machines like these nowadays, but the concept is similar from a virtual standpoint. On our server, we're using our keyboard to send input to a device that then displays output to another device. In our case, the input device is our keyboard and the output device is our screen, which is either connected directly to our server or is located on our computer which is connected to our server over a service such as SSH. On a Linux system, most processes run on a TTY, which is (to all intents and purposes) a terminal that grabs input and manages the output, similar to a teletypewriter in a virtual sense. A terminal is our method of interacting with our server.

In *Figure 7.3*, we have two processes running on a TTY of `tty1`, and the other processes are running on `pts/0`. The TTY we see is the actual terminal device, and `pts` references a virtual (pseudo) terminal device. Our server is actually able to run several `tty` sessions, typically one to seven. Each of these can be running their own programs and processes. To understand this better, try pressing *Ctrl* + *Alt* + any function key, from *F1* through *F7* (if you have a physical keyboard plugged into a physical server). Each time, you should see your screen cleared and then moved to another terminal. Each of these terminals is independent of one another. Each of your function keys represents a specific TTY, so by pressing *Ctrl* + *Alt* + *F6*, you're switching your display to TTY 6.

Essentially, you're switching from TTY 1 through to TTY 7, with each being able to contain their own running processes. If you run `ps a` again, you'll see any processes you start on those TTYs show up in the output as a `tty` session, such as `tty2` or `tty4`. Processes that you start in a terminal emulator will be given a designation of `pts`, because they're not running in an actual TTY, but rather a pseudo-TTY.

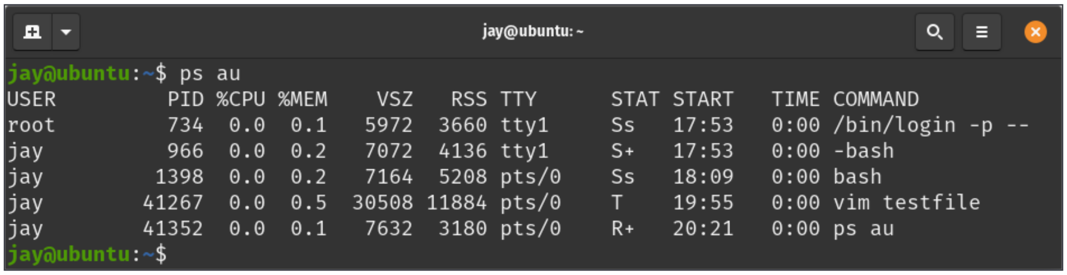
This was a long discussion for something that ends up being simple (TTY or pseudo-TTY), but with this knowledge you should be able to differentiate between a process running on the actual server or through a shell.

Continuing, let's take a look at the `TIME` field of our `ps` command output. This field represents the total amount of time the CPU has been utilized for that particular process. However, the time is `0:00` for each of the processes in the screenshot I've provided. This may be confusing at first. In my case, the `vim` processes in particular have been running for about 15 minutes or so since I took the screenshot, and they still show `0:00` utilization time even now. Actually, this isn't the amount of time the process has been running, but rather the amount of time the process has been actively engaging with the CPU. In the case of `vim`, each of these processes is just a buffer with a file open. For the sake of comparison, the Linux machine I'm writing this chapter on has a process ID of `759` with a time of `92:51`. PID `759` belongs to my X server, which provides the foundation for my **graphical user interface** (**GUI**) and windowing capabilities. However, this laptop currently has an uptime of 6 days and 22 hours as I type this, which is roughly equivalent to 166 hours, which is not the same amount of time that PID `759` is reporting in its `TIME` entry. Therefore, we can deduce that even though my laptop has been running 6 days straight, the X server has only utilized 92 hours and 51 minutes of actual CPU time. In summary, the `TIME` column refers to the amount of time a process needs the CPU in order to calculate something and is not necessarily equal to how long something has been running, or for how long a graphical process is showing on your screen.

Let's continue on with the `ps` command and look at some additional options. First, let's see what we get when we add the `u` option to our previous example, which gives us the following example command:

```
ps au
```

This will produce an output that will look similar to the following:



```
jay@ubuntu:~$ ps au
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         734  0.0  0.1   5972  3660 tty1     Ss   17:53   0:00 /bin/login -p --
jay          966  0.0  0.2   7072  4136 tty1     S+   17:53   0:00 -bash
jay         1398  0.0  0.2   7164  5208 pts/0    Ss   18:09   0:00 bash
jay        41267  0.0  0.5  30508 11884 pts/0    T    19:55   0:00 vim testfile
jay        41352  0.0  0.1   7632  3180 pts/0    R+   20:21   0:00 ps au
jay@ubuntu:~$
```

Figure 7.4: The output of the ps au command

When you run it, you should notice the difference from the `ps a` command right away. With this variation, you'll see processes listed that are being run by your user ID, as well as other users. When I run it, I see processes listed in the output for my user (`jay`), as well as one for `root`. The `u` option will be a common option you're likely to use, since most of the time while managing servers, you're probably more interested in keeping an eye on what kinds of shenanigans your users are getting themselves into. But perhaps the most common use of the `ps` command is the following variation:

```
ps aux
```

With the `x` option added, we're no longer limiting our output to processes within a TTY (either native or pseudo). The result is that we'll see a lot more processes, including system-level processes that are not tied to a process we started ourselves. Go ahead and try it. In practice, though, the `ps aux` command is most commonly used with `grep` to look for a particular process or string. For example, let's say you want to see a list of all `nginx` worker processes. To do that, you may execute a command such as the following:

```
ps aux | grep nginx
```

Here, we're executing the `ps aux` command as before, but we're piping the output into `grep`, where we're looking only for lines of output that include the string `nginx`. In practice, this is the way I often use `ps`, as well as the way I've noticed many other administrators using it. With `ps aux`, we are able to see a lot more output, and then we can narrow that down with search criteria by piping into `grep`. However, if all we wanted to do was to show processes that have a particular string, we could also do the following:

```
ps u -C nginx
```

This would produce output containing a list of processes matching `nginx`, and related details. Another useful variation of the `ps` command is to sort the output by sorting the processes using the most CPU first:

```
ps aux --sort=-pcpu
```

Unfortunately, that command shows a lot of output, and we would have to scroll back to the top in order to see the top processes. Depending on your terminal, you may not have the ability to scroll back very far (or at all), so the following command will narrow it down further:

```
ps aux --sort=-pcpu | head -n 5
```

Now that is useful! With that example, I'm using the `ps aux` command with the `--sort` option, sorting by the percentage of CPU utilization (`-pcpu`). Then I'm piping the output into the `head` command, where I'm instructing it to show me only five lines (`-n 5`). Essentially, this is giving me a list of the top five processes that have used the most CPU since boot time. In fact, I can do the same, but with the most-used memory instead:

```
ps aux --sort=-pmem | head -n 5
```

If you want to determine which processes are misbehaving and using a non-ordinary amount of memory or CPU, those commands will help you narrow it down. The `ps` command is a very useful command for your admin toolbox. Feel free to experiment with it beyond the examples I've provided; you can consult the man pages for the `ps` command to learn even more tricks. In fact, the second section of the man page for `ps` (under examples) gives you even more neat examples to try out.

Now that we know how to inspect running processes, in the next section we'll take a look at how to change the priority of the processes to ensure those that are more important are given extra attention by the CPU.

## Changing the priority of processes

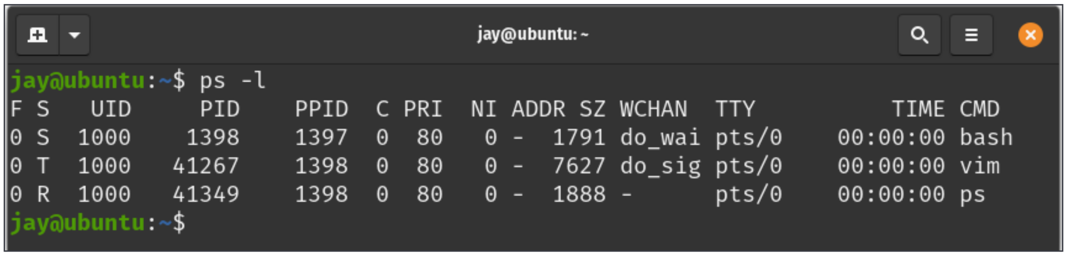
Processes on a Linux system can be run with an altered priority, giving some processes more priority and others less. This gives you, the administrator, full reign when it comes to ensuring that the most important processes on the system are running with an adequate level of prioritization. There are dedicated commands for this purpose: `nice` and `renice`. These commands allow you to launch a process with a specific priority, or change the priority of a process that's already running.

Nowadays, manually editing the priority of a process is something administrators will find themselves doing less often than they used to. A processor with 32 cores (or many more) is not all that uncommon, and neither is hundreds of gigabytes of RAM. Servers nowadays are certainly more powerful than they used to be, and are nowhere near as resource-starved as machines of old. Many servers (such as virtual machines) and containers are dedicated to a single task, so process tuning may not be of extreme value anymore. However, data processing firms and companies utilizing deep learning functions may find themselves needing to fine-tune some things.

Regardless of whether or not prioritizing processes is something that will be immediately useful to you, it's a good idea to at least understand the concept just in case you do find yourself needing to increase or decrease the priority of a process some day. Let's revisit the `ps` command, this time with the `-l` argument:

```
ps -l
```

The output of this command will appear as follows:



```
jay@ubuntu:~$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1398    1397  0  80   0 -  1791 do_wai pts/0    00:00:00 bash
0 T  1000   41267    1398  0  80   0 -  7627 do_sig pts/0    00:00:00 vim
0 R  1000   41349    1398  0  80   0 -  1888 -      pts/0    00:00:00 ps
jay@ubuntu:~$
```

Figure 7.5: The output of the ps -l command

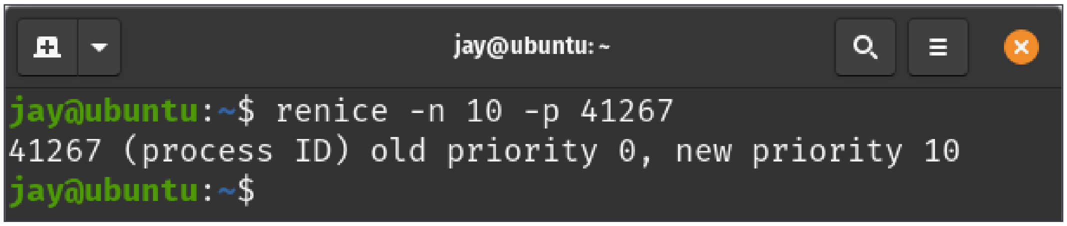
With the output of the `ps -l` command, notice the `PRI` and `NI` columns. `PRI` refers to the priority, and `NI` pertains to the "niceness" value, which we'll discuss in more detail later in this section. In this example, each process that I'm running has a `PRI` of `80`, and an `NI` of `0`. I didn't change or alter any of these; these are the values that I get when I start processes with no special tweaks. A `PRI` value of `80` is the starting value for that value on all processes, and will change as we increase or decrease the niceness value.

As I mentioned, we have dedicated commands that allow us to alter priorities, `nice` and `renice`. To determine which to use, it all comes down to whether or not the process is already running. With regard to the processes listed in *Figure 7.5*, we would want to use `renice` to change the priority for those, since they're all already running. If we wanted to launch a process with a specific priority right from the beginning, we would use `nice` instead.

For example, let's change the process of the `vim` session I have running. Sure, this is a somewhat lame example, as `vim` isn't a very important process. In the real world, you'd be prioritizing processes that are actually important. In my case, since the `vim` process has a PID of `41267`, the command I would need to run in order to change the niceness would become this:

```
renice -n 10 -p 41267
```

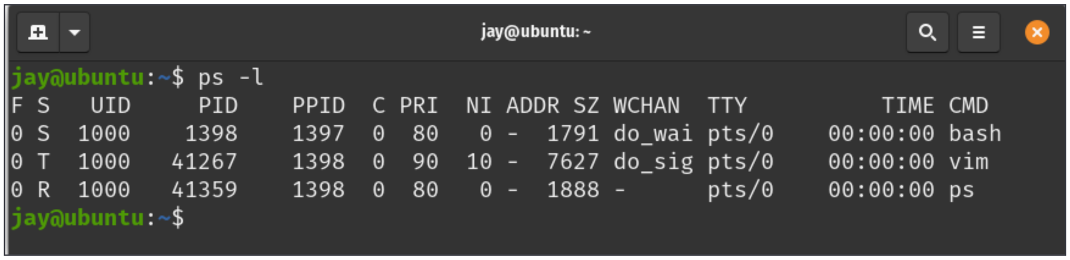
The output of this command will appear as follows:



```
jay@ubuntu:~$ renice -n 10 -p 41267
41267 (process ID) old priority 0, new priority 10
jay@ubuntu:~$
```

Figure 7.6: Changing the priority of a process with renice

If we run `ps -l` again, we can see the new nice value for `vim`:


jay@ubuntu: ~

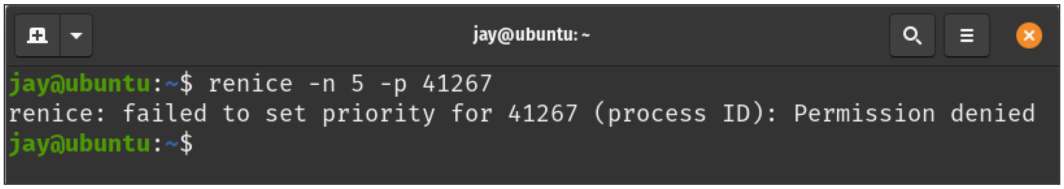
```
jay@ubuntu:~$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1398    1397  0  80   0 -  1791 do_wai pts/0    00:00:00 bash
0 T  1000   41267    1398  0  90  10 -  7627 do_sig pts/0    00:00:00 vim
0 R  1000   41359    1398  0  80   0 -  1888 -      pts/0    00:00:00 ps
jay@ubuntu:~$
```

Figure 7.7: The output of the ps -l command after changing the priority of a process

The new nice value of `10` now shows up for `vim` under `NI`, and the `PRI` value has increased to `90`. Now, this instance of `vim` will run at a lower priority than my other tasks, the reason being that the higher the nice value, the lower the priority. Notice that I didn't use `sudo` with the command when I changed the priority. In this example, that's okay because I'm increasing the nice value of the process, and that's allowed. However, let's try to decrease the nice value without `sudo`, using the following command:

```
renice -n 5 -p 41267
```

As you can see in the following output, I won't be as successful:


jay@ubuntu: ~

```
jay@ubuntu:~$ renice -n 5 -p 41267
renice: failed to set priority for 41267 (process ID): Permission denied
jay@ubuntu:~$
```

Figure 7.8: Attempting to decrease the priority of a process

My attempt to decrease the nice value from 10 down to 5 was blocked. If I were able to lower the niceness, then my process would be running at a higher priority. Instead, I received a `Permission denied` error. So essentially, users are allowed to increase the niceness of their processes, but are not allowed to decrease it, not even for processes they've initiated themselves. If you wish to decrease the nice value, you'll need to do so with `sudo`. So essentially, if you want to be "nicer," you can go ahead and do so. If you wish to be "meaner," you'll need `root` privileges. In addition, a user won't be able to change the priority of a process they don't own. So, if you attempt to use `renice` to change the niceness of a task running as a different user, you'll receive an `Operation not permitted` error.

At this point, we know how to re-prioritize our running processes with `renice`. Now, let's take a look at starting a new process with a specific priority with `nice`. Consider the following command:

```
nice -n 10 vim
```

Here, we're launching a new instance of `vim`, but with the priority set to a specific value right from the start. If we want to change the priority of `vim` again later, we'll need to use `renice`. As I mentioned earlier, `nice` is used to launch a new process with a specific priority, and `renice` is for changing the priority of a pre-existing process. In this example, we launched `vim` and set its nice value to `10` in one command.

Changing the priority of a text editor such as `vim` may seem like an odd choice for a test case, and it is. But the `vim` editor is harmless, as the likelihood of us changing the priority of it leading to a system halt is extremely minimal. There's no practical reason I can think of where it would be useful to re-prioritize something like a text editor. The takeaway, though, is that you *can* change the priority of the processes running on your server. On a real server, you may have an important process that runs and generates a report, and that report must be delivered on time. Or perhaps you have a process that generates an export of data that a client needs to have in order to make an on-time deliverable. So, if you think of the bigger picture, you can replace `vim` with the name of a process that is actually important for you or your organization.

You might be wondering what "nice" means in the context of the `nice` and `renice` commands. The "nice" number essentially refers to how nice a process is to other users. The higher the nice value, the lower the priority. So, a value of 20 is nicer than a value of 10. In that case, processes with a niceness of 20 are running at a lower priority, and so are kinder to the other processes on the system. The niceness can range from -20 to 19. A process with a nice value of -20 is the highest priority possible, while 19 is the lowest priority it can have. The entire system is quite a bit more complicated than this simple description. Although I refer to the nice value as the priority, it actually isn't. The nice value is used to calculate the actual priority. But for now, if we simplify the nice value to be representative of the priority, and the nice value to equate to a lower priority, the higher the number gets, that's enough for now.

So far, we've been using the `nice` and `renice` commands along with the `-n` option to set the nice values directly. It may be interesting to note though that you can simplify the `renice` command and leave out the `-n` option:

```
renice 10 42467
```

That command sets the nice value of the process to a positive 10, similar to our other examples. We can also use a negative number for the niceness if we want to increase the priority:

```
sudo renice -10 42467
```

Although it doesn't save us much typing to leave out the `-n` option, now you know that it is a possibility. The other difference with that example was that I needed to use `sudo` since I'm decreasing the nice value (more on that later).

When it comes to the `nice` command, we can also leave out the `-n` option, but the command works a bit differently in this regard. The following won't work:

```
nice 15 vim
```

The syntax of `nice` is a bit different, so giving it a positive number directly won't work as it does with `renice`. For that, we'll need to add a hyphen in front:

```
nice -15 vim
```

When you look at that command, you may assume we're applying a negative number. Actually, that's not true. Since the syntax is different with `nice`, the `-15` value we used results in a positive 15. We needed the hyphen in front of the value to signify to `nice` that we're applying a value as an option. If we actually do want to use a negative value with `nice` while also avoiding the `-n` option, we would need to use two hyphens:

```
nice --10 vim
```

The difference with syntax between the two commands with the `-n` option is a bit confusing in my opinion, so I recommend simply using the `-n` option with `nice` and `renice`, as that's going to be more uniform between them:

```
nice -n 10 vim
sudo nice -n -10 vim
renice -n 10 42467
sudo renice -n -10 42467
```

Those examples show both `nice` and `renice` using the `-n` option, and setting both positive and negative values. Since the `-n` option is used the same way between the two commands, it may be easier to focus on committing that to memory rather than focusing on the specifics. As previously discussed, I used `sudo` with commands that set a negative value for niceness, since only `root` can change a process to, or start a process with, a niceness below `0`. You'll receive the following error if you try to do it anyway:

```
nice: cannot set niceness: Permission denied
```

This type of protection is somewhat important, because you may have some users who feel as though their processes are the most important, and try to prioritize them all the way to `-19`. At the end of the day, it's better for a system administrator to make decisions on which processes are allowed to reach a niceness value in the negative.

As an administrator of Ubuntu servers, it's up to you to decide which processes should be running, and at what priority. You'll then determine the best way to achieve the exact system state that's appropriate, and tuning process priority may be a part of that. If nothing else, learning the `nice` and `renice` commands gives you another utility for your toolset.

## Dealing with misbehaving processes

Regarding the `ps` command, by this point you know how to display processes running on your server, as well as how to narrow down the output by string or resource usage. But what can you actually do with that knowledge? As much as we hate to admit it, sometimes the processes our server runs fail or misbehave and you need to restart them. If a process refuses to close normally, you may need to kill that process. In this section, we introduce the `kill` and `killall` commands to serve that purpose.

The `kill` command accepts a PID as an argument and attempts to close a process gracefully. In a typical workflow where you need to terminate a process that won't do so on its own, you will first use the `ps` command to find the PID of the culprit. Then, knowing the PID, you can attempt to `kill` the process. For example, if PID `31258` needed to be killed, you could execute the following:

```
sudo kill 31258
```

If all goes well, the process will end. You can restart it or investigate why it failed by perusing its logs.

To better understand what the `kill` command does, you first will need to understand the basics of **Linux Signals**. Signals are used by both administrators and developers and can be sent to a process either by the kernel, another process, or manually with a command. A signal instructs the process of a request or change, and in some cases, to completely terminate. An example of such a signal is `SIGHUP`, which tells processes that their controlling terminal has exited. One situation in which this may occur is when you have a terminal emulator open, with several processes inside it running. If you close the terminal window (without stopping the processes you were running), they'll be sent the `SIGHUP` signal, which basically tells them to quit (essentially, it means the shell quit or hung up).

Other examples include `SIGINT` (where an application is running in the foreground and is stopped by pressing *Ctrl* + *c* on the keyboard) and `SIGTERM`, which, when sent to a process, asks it to cleanly terminate. Yet another example is `SIGKILL`, which forces a process to terminate uncleanly. In addition to a name, each signal is also represented by a value, such as `15` for `SIGTERM` and `9` for `SIGKILL`. Going over each of the signals is beyond the scope of this chapter (the advanced topics of signals are mainly only useful for developers), but you can view more information about them by consulting the man page if you're curious:

```
man 7 signal
```

For the purposes of this section, the two types of signals we are most concerned about are `SIGTERM(15)` and `SIGKILL(9)`. When we want to stop a process, we send one of these signals to it, and the `kill` command allows us to do just that. By default, the `kill` command sends signal `15` (`SIGTERM`), which tells the process to cleanly terminate. If successful, the process will free its memory and gracefully close. With our previous example `kill` command, we sent signal `15` to the process, since we didn't clarify which signal to send.

Terminating a process with `SIGKILL(9)` is considered an extreme last resort. When you send signal `9` to a process, it's the equivalent of ripping the carpet out from underneath it or blowing it up with a stick of dynamite. The process will be force-closed without giving it any time to react at all, so it's one of those things you should avoid using unless you've literally tried absolutely everything you can think of. In theory, sending signal `9` can cause corrupted files, memory issues, or other shenanigans to occur. As for me, I've never actually run into long-term damage to software from using it, but theoretically it can happen, so you want to only use it in extreme cases. One case where such a signal may be necessary is regarding `defunct` or **Zombie process** in a situation where they don't close on their own. These processes are basically dead already and are typically waiting on their parent processes to reap them.

If the parent process never attempts to do so, they will remain on the process list. This in and of itself may not really be a big issue, since these processes aren't technically doing anything. But if their presence is causing problems and you can't kill them, you could try to send `SIGKILL` to the process. There should be no harm in killing a zombie process, but you would want to give them time to be reaped first.

To send signal `9` to a process, you would use the `-9` option of the `kill` command. It should go without saying, though, to make sure you're executing it against the proper process ID:

```
sudo kill -9 31258
```

Just like that, the process with a PID of `31258` will vanish without a trace. Anything it was writing to will be in limbo, and it will be removed from memory instantly. If, for some reason, the process still manages to stay running (which is extremely rare), you probably would need to reboot the server to get rid of it, which is something I've only seen in a few, very rare cases. An example of this is a zombie process, which is a process that runs but isn't impacted by having signals sent to it, since such a process won't be scheduled for CPU time anyway. When it all comes down to it, if `kill -9` doesn't get rid of the process, nothing will.

Another method of killing a process is with the `killall` command, which is probably safer than the `kill` command (if for no other reason than there's a smaller chance you'll accidentally kill the wrong process). Like `kill`, `killall` allows you to send `SIGTERM` to a process, but unlike `kill`, you can do so by name. In addition, `killall` doesn't just kill one process, it kills any process it finds with the name you've given it as an option. To use `killall`, you would simply execute `killall` along with the name of a process:

```
sudo killall myprocess
```

Just like the `kill` command, you can also send signal `9` to the process as well:

```
sudo killall -9 myprocess
```

Again, use that only when necessary. In practice, though, you probably won't use `killall -9` very often (if ever), because it's rare for multiple processes under the same process name to become locked. If you need to send signal `9`, stick to the `kill` command if you can.

The `kill` and `killall` commands can be incredibly useful in the situation of a stuck process, but these are commands you would hope you don't have to use very often. Stuck processes can occur in situations where applications encounter a situation from which they can't recover, so if you constantly find yourself needing to kill processes, you may want to check for an update to the package responsible for the service or check your server for hardware issues.

In the next section, let's take a look at system processes that run in the background and provide a service to us or our users, such as a web server process or DHCP server.

## Managing system processes

System processes, also known as **daemons**, are programs that run in the background on your server and are typically started automatically when it boots. We don't usually manage these services directly as they run in the background to perform their duty, with or without needing our input. For example, if our server is a DHCP server and runs the `isc-dhcp-server` process, this process will run in the background, listening for DHCP requests and providing new IP assignments to them as they come in. Most of the time, when we install an application that runs as a service, Ubuntu will configure it to start when we boot our server, so we don't have to start it ourselves. Assuming the service doesn't run into an issue, it will happily continue performing its job forever until we tell it to stop. In Linux, services are managed by its `init` system, also referred to as `PID 1` since the `init` system of a Linux system always receives that PID.

In recent years, the way in which processes are managed in Ubuntu Server has changed considerably. Ubuntu has switched to `systemd` for its `init` system, which was previously Upstart until a few years ago. Ubuntu 16.04 was the first LTS release of Ubuntu with `systemd`, and this is also the case today in Ubuntu 20.04, since we've had two LTS releases since then. Since `systemd` has been the standard for quite some time now, we'll focus our attention on the commands used with it to manage our services. Older `init` systems are ageing out.

With `systemd`, services are known as **units**, but to all intents and purposes, the terms "service," "daemon," and "unit" all essentially mean the same thing. Since I started using Linux over 18 years ago, I still refer to `systemd` units as services, out of habit. To help us manage these "units," `systemd` includes the `systemctl` command, which allows you to start, stop, and view the status of units on our server. To help illustrate this, I'll use OpenSSH as an example.

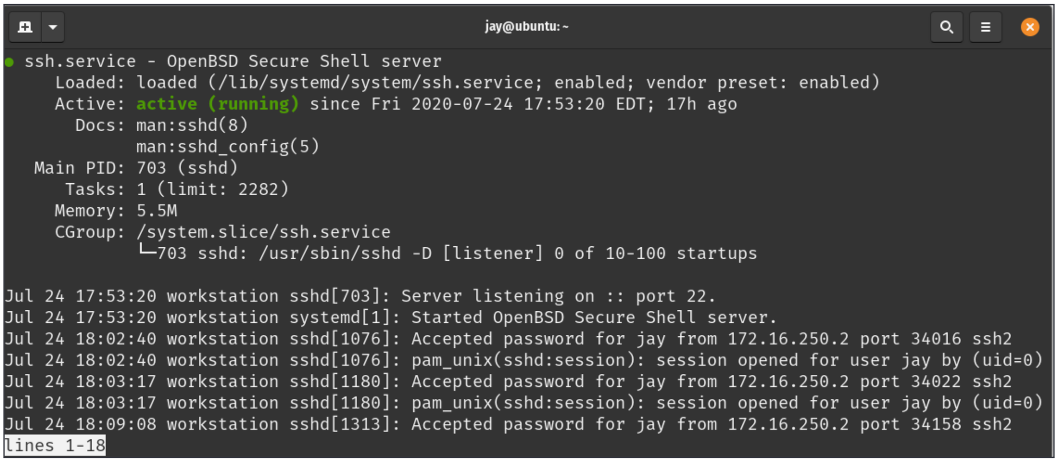
The name of the unit doesn't really matter, as the syntax of the `systemctl` command is the same regardless of the name of the unit we're interacting with. You can use `systemctl` to start, stop, or restart your Apache instance, your database server, or even use it to restart the entire networking stack. The `systemctl` command, with no options or parameters, assumes the `list-units` option, which dumps a list of units to your shell. This can be a bit messy, though, so if you already know the name of a unit you'd like to search for, you can pipe the output into `grep` and search for a string. This is handy in a situation where you may not know the exact name of the unit, but you know part of it:

```
systemctl |grep ssh
```

If you want to check the health of a unit, the best way is to actually use the `status` keyword, which will show you some very useful information regarding the unit. This information includes whether or not the unit is running, if it's enabled (meaning it's configured to start at boot time), as well as the most recent log entries for the unit:

```
systemctl status ssh
```

This command will produce an output something like the following:



```
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
     Active: active (running) since Fri 2020-07-24 17:53:20 EDT; 17h ago
       Docs: man:sshd(8)
             man:sshd_config(5)
   Main PID: 703 (sshd)
      Tasks: 1 (limit: 2282)
     Memory: 5.5M
     CGroup: /system.slice/ssh.service
             └─703 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups

Jul 24 17:53:20 workstation sshd[703]: Server listening on :: port 22.
Jul 24 17:53:20 workstation systemd[1]: Started OpenBSD Secure Shell server.
Jul 24 18:02:40 workstation sshd[1076]: Accepted password for jay from 172.16.250.2 port 34016 ssh2
Jul 24 18:02:40 workstation sshd[1076]: pam_unix(sshd:session): session opened for user jay by (uid=0)
Jul 24 18:03:17 workstation sshd[1180]: Accepted password for jay from 172.16.250.2 port 34022 ssh2
Jul 24 18:03:17 workstation sshd[1180]: pam_unix(sshd:session): session opened for user jay by (uid=0)
Jul 24 18:09:08 workstation sshd[1313]: Accepted password for jay from 172.16.250.2 port 34158 ssh2
lines 1-18
```

Figure 7.9: Checking the status of a unit with systemctl

Most of the time, you can actually check the status of units without needing `root` access, but you may not see all the information available. In the screenshot, you can see several log entries for the `ssh` service, but some units do not show those entries without `sudo`. With the `ssh` unit in particular, we see the log entries when checking the status with or without `sudo`.

Another thing you may notice in the screenshot is that the name of the ssh unit is actually ssh.service, but you don't need to include the .service part of the name, since that is implied by default. Sometimes, while viewing the status of a process with systemctl, the output may be condensed to save space on the screen. To avoid this and see the full log entries, add the -l option:

```
systemctl status -l ssh
```

Another thing to pay attention to is the vendor preset of the unit. Most packages in Ubuntu that include a service file for systemd will enable it automatically, but other distributions typically don't start and enable units by default (such as CentOS). In the case of the ssh example, you can see that the vendor preset is set to enabled. This means that once you install the openssh-server package, the ssh.service unit will automatically be enabled. You can confirm this by checking the Active line (where the example output says active (running)), which tells us that the unit is running. The Loaded line clarifies that the unit is enabled, so we know that the next time we start the server, ssh will be loaded automatically. Although systemd units are typically enabled and started automatically in Ubuntu when installing their package, this can still vary. When you install a new package, make sure you check the status of the unit whenever you install a new package, so you'll be aware of its settings.

Starting and stopping a unit is just as easy; all you have to do is change the keyword you use with systemctl to start or stop in order to have the desired effect:

```
sudo systemctl stop ssh
sudo systemctl start ssh
```

There are additional keywords, such as restart (which takes care of the previous two command examples at the same time), and some units even feature reload, which allows you to activate new configuration settings without needing to bring down the entire application. An example of why this is useful is with Apache, which serves web pages to local or external users. If you stop Apache, all users will be disconnected from the website you're serving. If you add a new site, you can use reload rather than restart, which will activate any new configuration you may have added without disturbing the existing connections. We'll take a look at Apache in *Chapter 14*, *Serving Web Content*, so don't worry too much about Apache right now. It's just a good example of a unit with additional functionality. Not all units feature a reload option, so you should check the documentation of the application that provides the unit to be sure.

Since I mentioned starting and stopping the unit for OpenSSH in the previous examples, an interesting aside is that doing so will not disconnect a current SSH session to the server, should you have one open. If you stop the `ssh` service, it won't drop your connection. Open connections are maintained, and stopping SSH only prevents new connections from happening. Therefore, SSH is different when compared to other units (such as Apache) in that existing connections aren't dropped when restarting the unit.

As I mentioned before, if you want a unit to automatically start when the server boots, the unit will need to be enabled. Units are automatically enabled most of the time, but in case you find one that isn't enabled, you can enable it with the `enable` keyword:

```
sudo systemctl enable ssh
```

It's just as easy to disable a unit as well:

```
sudo systemctl disable ssh
```

You can combine the process of not only enabling a unit, but also starting it at the same time:

```
sudo systemctl enable --now ssh
```

The `--now` argument tells `systemctl` to start the unit immediately after enabling it, rather than waiting for the next boot to do so or having you also run the `start` argument in a separate command.

Even though `systemd` is primarily used for managing units, it's actually an entire platform that manages multiple things on a Linux system, including DNS resolving, networking, and more. `systemd` even handles logging as well, and it provides us with the `journalctl` command, which we can use to view logging info (this is also why the output of `systemctl status ssh` was able to show us log entries). We've discussed logging a bit in *Chapter 4*, *Navigating and Essential Commands*, and we'll do so in more detail during *Chapter 21*, *Troubleshooting Ubuntu Servers* (which will also include further discussion of the `journalctl` command). For now, just understand that `systemd` is quite extensive when it comes to the number of things it helps us manage. For the purposes of this chapter, however, if you understand how to start, stop, enable, disable, and check the status of a unit, you're good to go for now.

## Scheduling tasks with cron

Earlier in this chapter, we worked through starting processes and enabling them to run in the background, and ensuring they start as soon as the server boots. In some cases, you may need an application to perform a job at a specific time, rather than to have it always running in the background. This is where **cron** comes in. With `cron`, you can set a process, program, or script to run at a specific time, down to the minute. Each user is able to have their own set of `cron` configuration (known as a `crontab`), which can perform any function that a user would be able to do normally. The `root` user has a `crontab` as well, which allows system-wide administrative tasks to be performed. Each `crontab` includes a list of `cron` jobs (one per line), which we'll get into shortly. To view a `crontab` for a user, we can use the `crontab` command:

```
crontab -l
```

With the `-l` option, the `crontab` command will show you a list of jobs for the user who executed the command. If you execute it as `root`, you'll see the root account's `crontab`. If you execute it as user `jdoe`, you'll see the `crontab` for `jdoe`, and so on. If you want to view a `crontab` for a user other than yourself, you can use the `-u` option and specify a user, but you'll need to execute it as `root` or with `sudo` to view the `crontab` for someone other than the user you're logged in as:

```
sudo crontab -u jdoe -l
```

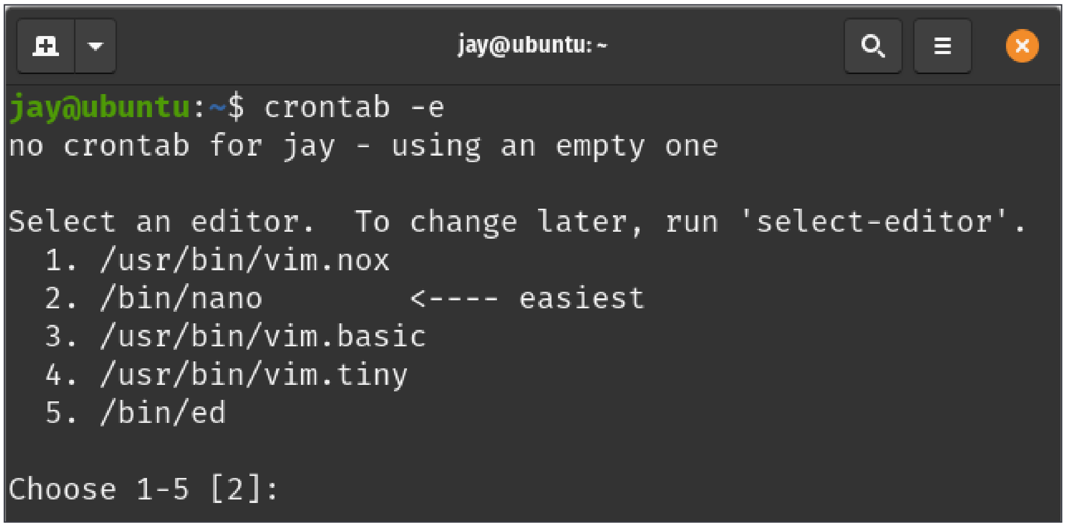
By default, no user has a `crontab` until you create one or more jobs. Therefore, you'll probably see output such as the following when you check for your current users:

```
no crontab for jdoe
```

To create a `cron` job, first log in as the user account you want the task to run under. Then, issue the following command:

```
crontab -e
```

If you have more than one text editor on your system, you may see output similar to the following:



```
jay@ubuntu:~$ crontab -e
no crontab for jay - using an empty one

Select an editor.  To change later, run 'select-editor'.
  1. /usr/bin/vim.nox
  2. /bin/nano        <---- easiest
  3. /usr/bin/vim.basic
  4. /usr/bin/vim.tiny
  5. /bin/ed

Choose 1-5 [2]:
```

Figure 7.10: Selecting an editor for use with the crontab command

In this case, you'll simply press the number corresponding to the text editor you'd like to use when creating your `cron` job. To choose an editor and edit your `crontab` with a single command, the following command will do exactly that:

```
EDITOR=vim crontab -e
```

In this example, you can replace `vim` with whatever text editor you prefer. At this point, you should be placed in a text editor with your `crontab` file open. The default `crontab` file for each user features some helpful comments that give you some useful information regarding how `cron` works. To add a new job, you would scroll to the bottom of the file (after all the comments) and insert a new line. Formatting is very particular here, and the example comments in the file give you some clue as to how each line is laid out. Specifically, this part:

```
m h dom mon dow command
```

Each `cron` job has six fields, each separated by at least one space or tab spaces. If you use more than one space, or tab, `cron` is smart enough to parse the file properly. In the first field, we have the minute in which we would like the job to occur. In the second field, we place the hour in the 24-hour format, from 0-23. The third field represents the day of the month. In this field, you can place a 5 (5th of the month), 23 (23rd of the month), and so on. The fourth field corresponds to the month, such as 3 for March or 12 for December. The fifth field is the day of the week, numbered from 0-6 to represent Sunday through Saturday. Finally, in the last field, we have the command to be executed. A few example `crontab` lines are as follows:

```
3 0 * * 4 /usr/local/bin/cleanup.sh
* 0 * * * /usr/bin/apt update
0 1 1 * * /usr/local/bin/run_report.sh
```

With the first example, the `cleanup.sh` script, located in `/usr/local/bin`, will be run at 12:03 a.m. every Thursday. We know this because the minute column is set to `3`, the hour column is set to `0` (midnight), the day column is `4` (Thursday), and the command column shows a fully qualified command of `/usr/local/bin/cleanup.sh`.

What does it mean for a command to be *fully qualified*? Basically, a command being fully qualified means that the entire path to the binary responsible for the command is completely typed out. In the second example, we could have simply typed `apt update` for the command and that would've probably worked just fine. However, not including the full path to the program is considered bad `cron` etiquette. While the command may have worked without being fully qualified, its success would depend on the application being found in the path of the user who is calling it. Not all servers are set up the same, so this might not work depending on how the shell is set up. If you include the full path, the job should run regardless of how the underlying shell is configured.

If you don't know what the fully qualified command is, all you have to do is use the `which` command. This command, when used with the name of a command you'd like to run, will give you the fully qualified command if the command is located on your system.

Continuing with the second example, we're running `/usr/bin/apt update` to update our server's repository index every morning at midnight. The asterisks on each line refer to *any*, so with the minute column being simply `*`, that means that this task is eligible for any minute. Basically, the only field we clarified was the hour field, which we set to `0` in order to represent 12:00 a.m.

With the third example, we're running the `/usr/local/bin/run_report.sh` script on the first day of every month at 01:00 a.m. If you notice, we set the third column (**day of month**) to `1`, which is the same as February 1st, March 1st, and so on. This job will be run if it's the first day of the month, but only if the current time is also 01:00 a.m., since we filled in the first and second column, which represent the minute and hour, respectively.

Once you finish editing a user's `crontab` and save it, `cron` is updated and, from that point forward, will execute the task at the time you select. The `crontab` will be executed according to the current time and date on your server, so you want to make sure that that is correct as well, otherwise you'll have your jobs execute at unexpected times. You can view the current date and time on your server by simply issuing the `date` command.

To get the hang of creating jobs with `cron`, the best way (as always) is to practice. The second example `cron` job is probably a good one to experiment with, as updating your repository index isn't going to hurt anything.

## Summary

In this chapter, we learned how to manage processes. We began with a look at the `ps` command, which we can use to view a list of processes that are currently running. We also took a look at managing jobs, as well as killing processes that, for one reason or another, are misbehaving. We also discussed methods of changing the priority of a process, to ensure we have full control over which processes are given more processing time, and we also learned how we can schedule things to run at a later time and date with cron.

In *Chapter 8*, *Monitoring System Resources*, we'll take a look at some ways we can keep an eye on the resources that are available on our server, where we will learn how to check disk usage, understand memory usage and swap space, as well as a look at some utilities that can make resource management a breeze.

## Further reading

- Getting Started with tmux: `https://learnlinux.link/tmux`
- Ham Vocke, *A Quick and Easy Guide to tmux*: `http://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux/`
- Tmux Cheat Sheet and Quick Reference: `https://tmuxcheatsheet.com/`
- crontab guru: `https://crontab.guru/`

# 8
# Monitoring System Resources

In the last chapter, we learned how we can manage tasks that are running on our server. We now know how to see what's running in the background, how to enable or disable a unit from starting at boot time, and also how to schedule tasks to run in the future. But in order for us to be able to effectively manage the tasks that our servers carry out, we also need to keep an eye on system resources. If we run out of RAM, fill up our disk, or overload our CPU, then a server that normally processes tasks very efficiently might come to a screeching halt. In this chapter, we'll take a look at these resources and how to monitor them.

Our discussion on resource management will include:

- Viewing disk usage
- Monitoring memory usage
- Understanding load average
- Viewing resource usage with `htop`

One resource that is extremely important on our servers is storage, and keeping track of such things as available disk space is critical as even the most powerful server you can purchase would be unable to function without free disk space. We'll take a look at some ways to monitor disk usage in the next section.

## Viewing disk usage

Keeping an eye on your storage is always important, as no one enjoys getting a call in the middle of the night saying that a server encountered an issue, especially not something that could've been easily avoided, such as a filesystem growing too close to being full. Managing storage on Linux systems is simple once you master the related tools, the most useful of which I'll go over in this section. In particular, we'll look at tools we can use to answer the question "what's using up all the disk space?", which is the most common question that comes up when dealing with disk usage.

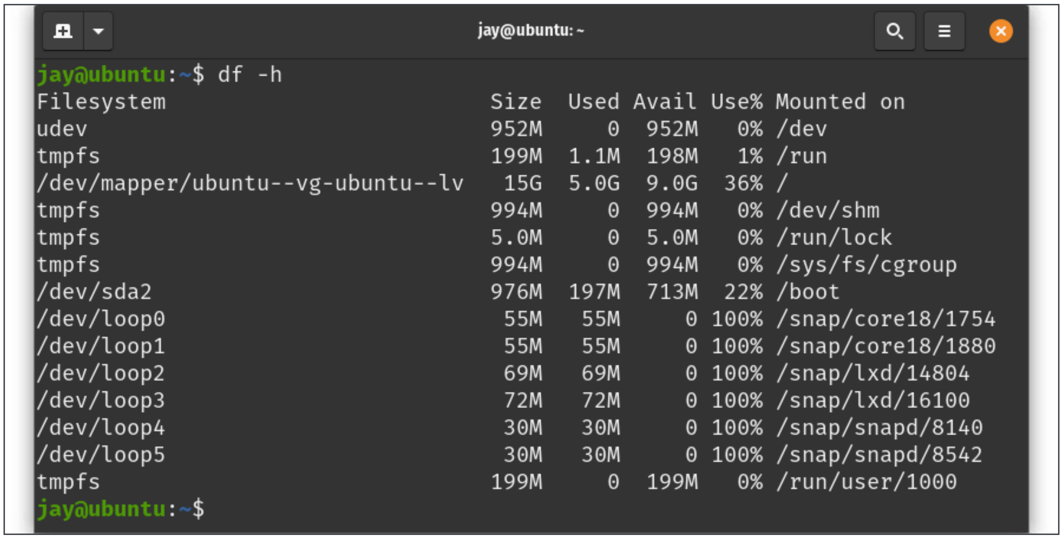
First, let's look at the `df` command.

### Using df

The `df` command is likely always going to be your starting point in situations where you don't already know which volume or mount point is becoming full. When executed, it gives you a high-level overview, so it's not necessarily useful when you want to figure out who or what in particular is hogging all your space. However, when you just want to list all your mounted volumes and see how much space is left on each, `df` fits the bill. By default, it shows you the information in bytes. However, I find it easier to use the `-h` option (which produces a human-readable output) with `df` so that you'll see information that's a bit easier to read. Go ahead and give it a try:

```
df -h
```

This should produce an output that looks something like the following:



```
jay@ubuntu:~$ df -h
Filesystem                         Size  Used Avail Use% Mounted on
udev                               952M     0  952M   0% /dev
tmpfs                              199M  1.1M  198M   1% /run
/dev/mapper/ubuntu--vg-ubuntu--lv   15G  5.0G  9.0G  36% /
tmpfs                              994M     0  994M   0% /dev/shm
tmpfs                              5.0M     0  5.0M   0% /run/lock
tmpfs                              994M     0  994M   0% /sys/fs/cgroup
/dev/sda2                          976M  197M  713M  22% /boot
/dev/loop0                          55M   55M     0 100% /snap/core18/1754
/dev/loop1                          55M   55M     0 100% /snap/core18/1880
/dev/loop2                          69M   69M     0 100% /snap/lxd/14804
/dev/loop3                          72M   72M     0 100% /snap/lxd/16100
/dev/loop4                          30M   30M     0 100% /snap/snapd/8140
/dev/loop5                          30M   30M     0 100% /snap/snapd/8542
tmpfs                              199M     0  199M   0% /run/user/1000
jay@ubuntu:~$
```

Figure 8.1: Output from the df -h command

The output will look different depending on the types of disks and mount points on your system. In the screenshot, you'll see that the root filesystem is located on `/dev/mapper/ubuntu--vg-ubuntu--lv`. We know this because under the column `Mounted on` we see that the mount point is set to a single forward slash (`/`). As we discussed in *Chapter 4*, *Navigating and Essential Commands*, this single forward slash refers to the beginning of the filesystem (also referred to as the root filesystem). In my case, this is an LVM volume, which is why we have a device with such a long name, beginning with `/dev/mapper`. Let's not worry about LVM for now, we'll discuss that later. But for now, just keep in mind that the single forward slash refers to the beginning of the filesystem, and the device name on the left refers to the actual device that's mounted there.

The actual device name varies from one server to another, and also varies depending on whether you chose to utilize LVM during installation. Instead of a long path beginning with `/dev/mapper`, you may instead see the device name as `/dev/sda1`, `/dev/xvda1`, `/dev/nvme0n1p1`, or other variations. The name of the device is generated by the type of hardware the underlying storage device is, such as the `/dev/nvme...` naming convention used for NVME hard drives, `/dev/sdaN` for standard SATA hard drives, and so on.

The actual type of device the underlying hardware is doesn't matter so much; it only really matters that you can identify which device is at the most danger of becoming full. In the example screenshot, the root filesystem is using `36%` of its available space. In this case, we aren't in danger of running out of space. There are some loopback devices that are up to `100%` usage (identified by devices with a naming scheme of `/dev/loopN`) but those aren't actually of concern, as system processes may create loopback devices for various purposes as needed.

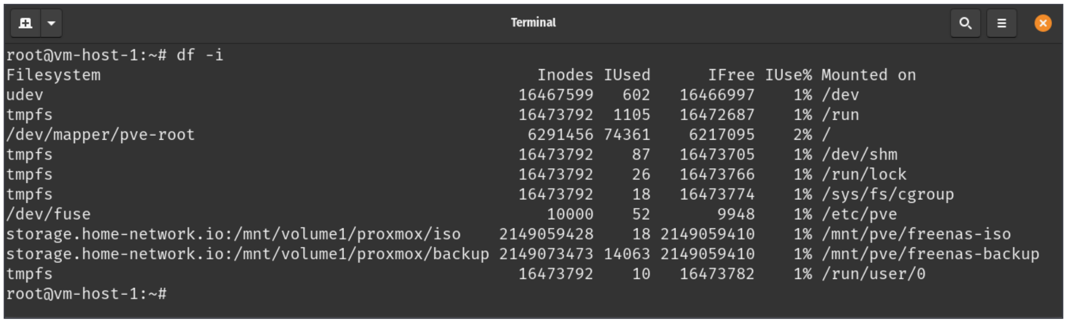
While investigating disk utilization, it's also important to check inode utilization as well. Checking inode utilization will be especially helpful in situations where it's being reported that your disk is full, yet the `df -h` command shows plenty of free space is available. It can definitely be very confusing the first time you run into this situation. In such a scenario, it may be that you've run out of inodes, and your disk isn't actually full from a free space perspective.

But, what exactly is an inode, and why would such a thing cause a disk to be reported as full when it's actually not? Think of the concept of an inode as a type of *database object*, containing metadata for the actual items you're storing. Information stored in inodes are details such as the owner of the file, permissions, last modified date, and type (whether it is a directory or a file). While metadata is certainly a good thing to have, the problem with inodes is that you can only have a limited number of them on any storage device.

This number is usually extremely high and the limit is very hard to reach, though. In the case of servers that are used for workloads that can result in your organization storing hundreds of thousands of files, the inode limit can become a real problem. I'll show you some output from one of my servers to help illustrate this:

```
df -i
```

The output of this command is as follows:



```
root@vm-host-1:~# df -i
Filesystem                                         Inodes IUsed      IFree IUse% Mounted on
udev                                             16467599   602   16466997    1% /dev
tmpfs                                            16473792  1105   16472687    1% /run
/dev/mapper/pve-root                              6291456 74361    6217095    2% /
tmpfs                                            16473792    87   16473705    1% /dev/shm
tmpfs                                            16473792    26   16473766    1% /run/lock
tmpfs                                            16473792    18   16473774    1% /sys/fs/cgroup
/dev/fuse                                           10000    52       9948    1% /etc/pve
storage.home-network.io:/mnt/volume1/proxmox/iso    2149059428    18 2149059410    1% /mnt/pve/freenas-iso
storage.home-network.io:/mnt/volume1/proxmox/backup 2149073473 14063 2149059410    1% /mnt/pve/freenas-backup
tmpfs                                            16473792    10   16473782    1% /run/user/0
root@vm-host-1:~#
```

Figure 8.2: Output from the df -i command

As you can see, the `-i` option of `df` gives us information regarding inodes instead of the actual space used. In this example, the root filesystem on the example server has a total of `6291456` inodes available, of which `74361` are used and `6217095` are free. If you have a system that's reporting a full disk (though you see plenty of space is free when running `df -h`), it may actually be an issue with your volume running out of inodes. In this case, the problem would not be the size of the files on your disk, but rather the sheer number of files you're storing. In my experience, I've seen this happen because of mail servers becoming bound (millions of stuck emails, with each email being a file), as well as unmaintained log directories. It may seem as though having to contend with an inode limitation is unbecoming of a legendary platform such as Linux, however, as I mentioned earlier, this limit is very hard to reach unless something is very, very wrong. So in summary, in a situation where you're getting an error message that no space is available and you have plenty of space, check the inodes.

## Diving deeper into disk usage

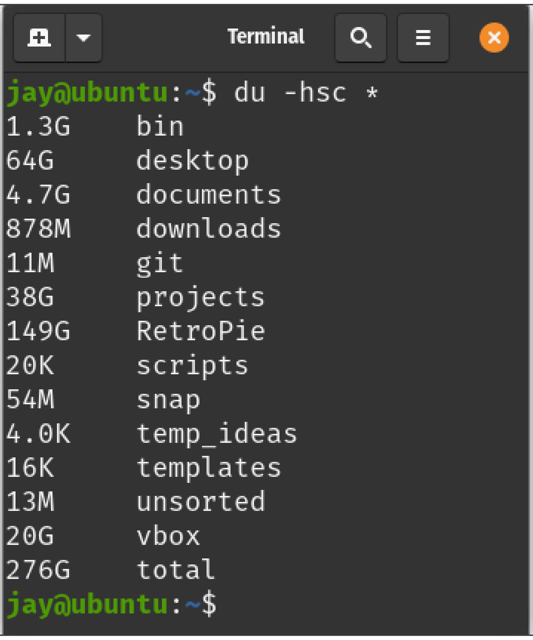
The next step in investigating what's gobbling up your disk space is finding out which files in particular are using it all up. At this stage, there is a multitude of tools you can use to investigate. The first I'll mention is the `du` command, which is able to show you how much space a directory is using. Using `du` against directories and sub-directories will help you narrow down the problem. Like `df`, we can also use the `-h` option with `du` to make our output easier to read. By default, `du` will scan the current working directory your shell is attached to and give you a list of each item within the directory, the total space each item consists of, as well as a summary at the end.

The `du` command is only able to scan directories that its calling user has permission to scan. If you run this as a non-root user, then you may not be getting the full picture. Also, the more files and sub-directories that are within your current working directory, the longer this command will take to execute. If you have an idea where the resource hog might be, try to `cd` into a directory further in the tree to narrow your search down and reduce the amount of time the command will take.

The output of `du -h` can often be more verbose than you actually need in order to pinpoint your culprit and can fill several screens. To simplify it, my favorite variation of this command is the following:

```
du -hsc *
```

Basically, you would run `du -hsc *` within a directory that's as close as possible to where you think the problem is. The `-h` option, as we know, gives us human-readable output (essentially, giving us output in the form of megabytes, gigabytes, and so on). The `-s` option gives us a summary and `-c` provides us with the total amount of space used within our current working directory. The following screenshot shows this output from my laptop:



```
jay@ubuntu:~$ du -hsc *
1.3G    bin
64G     desktop
4.7G    documents
878M    downloads
11M     git
38G     projects
149G    RetroPie
20K     scripts
54M     snap
4.0K    temp_ideas
16K     templates
13M     unsorted
20G     vbox
276G    total
jay@ubuntu:~$
```

Figure 8.3: Example output from du -hsc *

To make that example more interesting, I took the screenshot from my personal desktop, but the resulting command and its syntax are the same. As you can see, the information provided by `du -hsc *` is a nice, concise summary. From the output, we can clearly see how much space each of the directories within our working directory takes currently. For example, I have 38 GB used in my `projects` directory right now. Not only am I storing the files for this book in that folder, I'm also storing raw video recordings there for my YouTube channel, so this directory can become quite large at times.

At this point, we know which directories at the top level of our current working directories are using the most space. But we still need to narrow this down to *what* in particular within those directories is responsible for using that space. To dive deeper, we could `cd` into any of those large directories and run the `du` command again. After a few runs, we should be able to narrow down the largest files within these directories and make a decision on what we want to do with them. Perhaps we can clean unnecessary files or add another disk. Once we know what is using up our space, we can decide what we're going to do about it.

At this point in reading this book, you're probably under the impression that I have some sort of strange fixation on saving the best for last. You'd be right. I'd like to finish off this section by introducing you to one of my favorite applications, the **NCurses Disk Usage** utility (or more simply, the `ncdu`). The `ncdu` command is one of those things that administrators who constantly find themselves dealing with disk space issues learn to love and appreciate. In one go, this command gives you not only a summary of what is eating up all your space, it also gives you an ability to traverse the results without having to run a command over and over while manually traversing your directory tree. You simply execute it once and then you can navigate the results and drill down as far as you need.

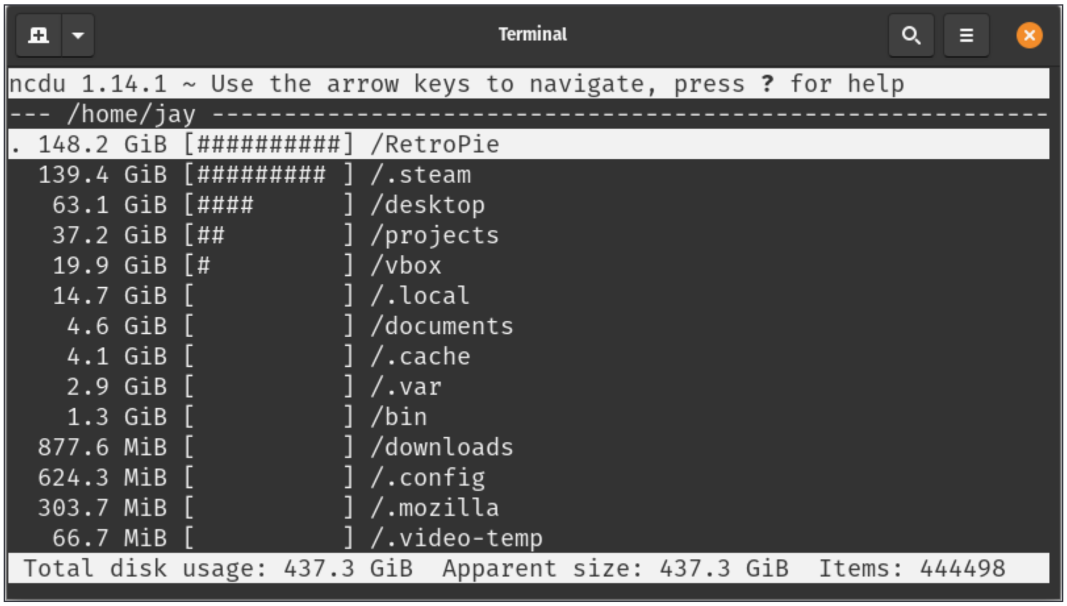
To use `ncdu`, you will need to install it as it doesn't come with Ubuntu by default:

```
sudo apt install ncdu
```

Once installed, simply execute `ncdu` in your shell from any starting directory of your choosing. When done, simply press *q* on your keyboard to quit. Like `du`, `ncdu` is only able to scan directories that the calling user has access to. You may need to run it as `root` to get an accurate portrayal of your disk usage.

> You may want to consider using the `-x` option with `ncdu`. This option will limit it to the current filesystem, meaning it won't scan network mounts or additional storage devices; it'll just focus on the device you started the scan on. This can save you from scanning areas that aren't related to your issue.

When executed, `ncdu` will scan every directory from its starting point onward. When finished, it will give you a menu-driven layout allowing you to browse through your results:



```
ncdu 1.14.1 ~ Use the arrow keys to navigate, press ? for help
--- /home/jay -----------------------------------------------------------
. 148.2 GiB [##########] /RetroPie
  139.4 GiB [######### ] /.steam
   63.1 GiB [####      ] /desktop
   37.2 GiB [##        ] /projects
   19.9 GiB [#         ] /vbox
   14.7 GiB [          ] /.local
    4.6 GiB [          ] /documents
    4.1 GiB [          ] /.cache
    2.9 GiB [          ] /.var
    1.3 GiB [          ] /bin
  877.6 MiB [          ] /downloads
  624.3 MiB [          ] /.config
  303.7 MiB [          ] /.mozilla
   66.7 MiB [          ] /.video-temp
 Total disk usage: 437.3 GiB  Apparent size: 437.3 GiB  Items: 444498
```

Figure 8.4: ncdu in action

Again, I took this screenshot from my desktop, from within my `home` directory. What `ncdu` does is show you your disk usage from your current directory down, and it will order the results by placing the items with the highest usage toward the top. To move around inside of `ncdu`, you do so by moving your selection (indicated with a long white highlight) with the up and down arrows on your keyboard. If you press *Enter* on a directory, `ncdu` switches to showing you the summary of that directory, and you can continue to drill down as far as you need. In fact, you can actually delete items and entire folders by pressing *d*. Therefore, `ncdu` not only allows you to find what is using up your space, it also allows you to take action as well!

Sometimes, it's obvious what's taking up space on a disk, and `ncdu` may not always be necessary. Generally speaking, you'll start out your investigation with `df -h`, to see which storage volume is the one that's running out of space. Then, you'll go into that directory and run another command, such as `du -hsc *`, to see which directory is using up the most space. If you don't immediately know from the output of `du` what the underlying issue is, then consider using a tool such as `ncdu` to dive down even deeper.

Although monitoring storage is critical, we also need to keep an eye on free memory. Next up, we'll take a look at how to monitor the memory of our server.

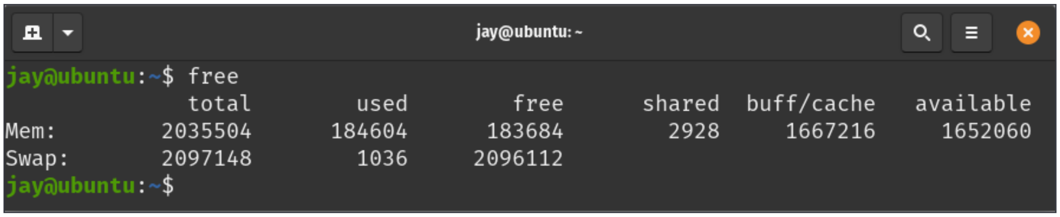
## Monitoring memory usage

I forget things all the time. I regularly forget where my car keys are, even though they're almost always right there in my pocket the entire time. I even forget to use `sudo` for commands that normally require it, over 18 years since starting to work with Linux. Thankfully, computers have a better memory than I do, but if we don't manage it effectively, the memory on our servers will be just as useless as I am when I forget to put freshly washed laundry in the dryer.

Understanding how Linux manages memory can actually be a somewhat complex topic, as understanding how much memory is truly free can be a hurdle for newcomers to overcome. You'll soon see that how Linux manages memory on your server is actually fairly straightforward once explained.

## Understanding server memory

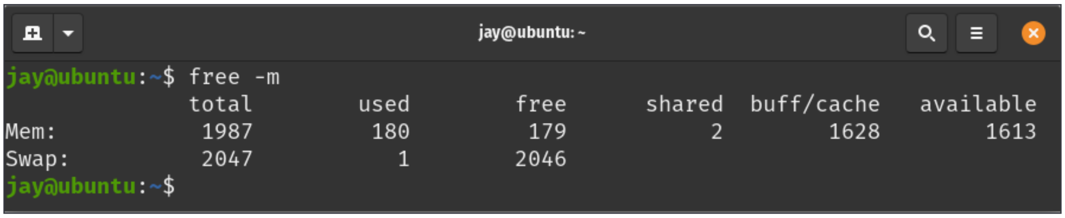
For the purpose of monitoring memory usage on our server, we have the `free` command at our disposal, which we can use to see how much memory is being consumed at any given time. Giving the `free` command with no options will result in the output being shown in terms of kilobytes:



```
jay@ubuntu:~$ free
              total        used        free      shared  buff/cache   available
Mem:        2035504      184604      183684        2928     1667216     1652060
Swap:       2097148        1036     2096112
jay@ubuntu:~$
```

Figure 8.5: Output of the free command

My favorite variation of this command is `free -m`, which shows the amount of memory in use in terms of megabytes. You can also use `free -g` to show the output in terms of gigabytes, but the output won't be precise enough on most servers. In my opinion, adding the `-m` option makes the `free` command much more readable:



```
jay@ubuntu:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:           1987         180         179           2        1628        1613
Swap:          2047           1        2046
jay@ubuntu:~$
```

Figure 8.6: Output of the free -m command

Since everything is broken down into megabytes, it's much easier to read, at least for me.

At first glance, it may appear as though this server has only `179` MB free. You'll see this in the first row and third column under `free`. In actuality, the number you'll really want to pay attention to is the number under `available`, which is `1613` MB in this case. That's now much memory is actually free. Since this server has `1987` MB of total RAM available (you'll see this on the first row, under `total`), this means that most of the RAM is free, and this server is not really working that hard at all.

Some additional explanation is necessary to truly understand these numbers. You could very well stop reading this section right now as long as you take away from it that the `available` column represents how much memory is free for your applications to use. However, it's not quite that simple. Technically, when you look at the output, the server really does have `179` MB free. The amount of memory listed under `available` is legitimately being used by the system in the form of a cache but would be freed up in the event that any application needed to use it. If an application starts and needs a decent chunk of memory in order to run, the kernel will provide it with some memory from this cache.

Linux, like most modern systems, subscribes to the belief that "unused RAM is wasted RAM." RAM that isn't being used by any process is given to what is known as a **disk cache**, which is utilized to make your server run more efficiently. When data needs to be written to a storage device, it's not directly written right away. Instead, this data is written to the disk cache (a portion of RAM that's set aside) and then synchronized to the storage device later in the background. The reason this makes your server more efficient is that this data being stored in RAM would be written to and retrieved faster than it would be from disk. Applications and services can synchronize data to the disk in the background without forcing you to wait for it. This cache also works for reading data, as when you first open a file, its contents are cached. The system will then retrieve it from RAM if you read the same file again, which is more efficient than loading it from the storage volume each time. If you just recently saved a new file and retrieve it right away, it's likely still in the cache and then retrieved from there, rather than from the disk directly.

To understand all of the columns shown in *Figure 8.6*, I'll outline the meaning of each in the following table:

| **Column** | **Meaning** |
|---|---|
| `total` | The total amount of RAM installed on the server. |
| `used` | The memory that is used (from any source). This is calculated as follows: *used = total - free - buffers - cache*. |
| `free` | The memory not being used by anything, the cache or otherwise. |
| `shared` | The memory used by `tmpfs` as well as other shared resources. |
| `buff/cache` | The amount of memory being used by the buffers and cache. |
| `available` | The memory that is free for applications to use. |

You may have noticed in *Figure 8.6* that memory usage is also listed for a resource called `Swap`. Let's take a look at that as well. We will dedicate the next section entirely to it so that we ensure we understand what it is, and what it does for us.

## Managing swap

**Swap** is one of those things we never want to use, but always want to make sure is available. It's kind of like car insurance, no one is excited to buy it, but we do want to have it in case something bad happens. There's even some debate between administrators on whether or not swap is still relevant today. It's definitely relevant, regardless of what anyone says, as it's a safety net of sorts. (And disk space is cheaper nowadays, so dedicating some of our storage to this task isn't really a big deal, so we may as well).

So what is it? Swap is basically a partition or a file that acts as RAM in situations where your server's memory is saturated. If we manage a server properly, we hope to never need it, as swap is stored on your hard disk, which is orders of magnitude slower than RAM. But if something goes wrong on your server and your memory usage skyrockets, swap may save you from having your server go down. The **Out of Memory (OOM) Killer** may also activate itself when memory is full, to kill a misbehaving process that's using the majority of your memory, but as much as possible, we don't want to rely on that and instead ensure adequate swap in case memory is exhausted.

The way swap is configured by default in Ubuntu has changed a bit since the time the first edition of this book was published. With Ubuntu 16.04 and earlier, a swap partition was automatically created for you if you chose the default partitioning scheme during installation, and if you didn't create a swap partition when partitioning your server, the installer would yell at you for it. However, a swap partition is no longer created by default in modern versions of Ubuntu, and the installer will create a `swap` *file* (rather than a partition) for you automatically. You may still see swap partitions on older installations, but going forward, this is the best way to handle it. At least until something better comes around. If you need more swap space, you can delete the `swap` file and recreate it. That's definitely an easier thing to do than having to resize your partition tables in order to enlarge swap, which is potentially dangerous if you make a mistake during the process.

Therefore, I'm not going to talk about creating a swap partition in this edition of the book, as there's no reason to do so anymore.

The `swap` file for your server is declared in the `/etc/fstab` file (we'll discuss the `/etc/fstab` file in more detail in *Chapter 9*, *Managing Storage Volumes*). In most cases, you would've had a `swap` file created for you during installation. You could, of course, add a swap partition later if for some reason you don't have one. In the case of some cloud instance providers, you may not get a `swap` file by default. In that situation, you would create a `swap` file yourself (we'll discuss the process later in this section) and then use the `swapon` command to activate it:

```
sudo swapon -a
```

When run, the `swapon -a` command will find your swap partition in`/etc/fstab` (if it's mentioned there), mount it, and activate it for use. The inverse of this command is `swapoff -a`, which deactivates your `swap` file. It's rare that you'd need to disable swap, unless, of course, you were planning on deleting your `swap` file in order to create a larger one. If you find out that your server has an inadequate swap partition size, that may be a course of action you would take.

While having swap is generally a good idea, there are actually some applications that prefer that the server doesn't have it at all. This is rare, but Kubernetes is a good example of this. If you're running Kubernetes, the installation of it will complain if you do have a swap partition. This is somewhat of a rare occurrence; not very many applications run better if the server has no `swap` file. In the case of a Kubernetes cluster, the individual servers within such a cluster would be a special case anyway, each dedicated to the task of running containers (which is what Kubernetes does; more on that in *Chapter 18*, *Container Orchestration*).

When you check your free memory (hint: execute `free -m`), you'll see `swap` listed whether you have it or not, but when swap is deactivated, you will see all zeros for the size totals.

So, how do you actually create a `swap` file? To do so, you'll first create the actual file to be used as swap. This can be stored anywhere, but `/swapfile` is typically ideal. You can use the `fallocate` command to create the actual file. The `fallocate` command will force a file to be a particular size:

```
sudo fallocate -l 4G /swapfile
```

Here, I'm creating a 4 GB `swap` file, but feel free to make yours whatever size you want in order to fit your needs. Next, we need to prepare this file to be used as swap. First, we'll need to fix the permissions as we need this file to be a bit more restrictive than most:

```
sudo chmod 0600 /swapfile
```

Then, we can use the `mkswap` command to convert this file into an actual `swap` file:

```
sudo mkswap /swapfile
```

Now, we have a handy-dandy `swap` file stored in our root filesystem. Next, we'll need to mount it. As always, it's recommended that we add this to our `/etc/fstab` file. What follows is an example entry:

```
/swapfile   none   swap   sw   0 0
```

From this point, we can activate our new `swap` file with the `swapon` command that I mentioned earlier:

```
sudo swapon -a
```

Now, the `swap` file is active and in use. While I certainly hope you won't need to resort to using swap, I know from experience that it's only a matter of time. Knowing how to add and activate swap when you need it is definitely a good practice, but for the most part, you should be fine because, by default on most platforms, you'll have swap created for you when setting up Ubuntu for the first time during installation. If you do need to create it manually for whatever reason, I always recommend a bare minimum of 2 GB on servers, but if you can manage to create a larger one for this purpose, that's even better.

How much swap is being used is something you should definitely keep an eye on. When the memory starts to get full, the server will start to utilize the `swap` file that was created during installation. It's normal for a small portion of swap to be utilized even when the majority of the RAM is free. But if a decent chunk of swap is being used, it should be investigated (perhaps a process is using a larger than normal amount of memory).

You can actually control at which point your server will begin to utilize swap. How frequently a Linux server utilizes swap is referred to as its swappiness. By default, the `swappiness` value on a Linux server is typically set to `60`. You can verify this with the following command:

```
cat /proc/sys/vm/swappiness
```

The higher the `swappiness` value, the more likely your server will utilize swap. If the `swappiness` value is set to `100`, your server will use swap as much as possible. If you set it to `0`, swap will never be used at all. This value correlates roughly to the percentage of RAM being used. For example, if you set `swappiness` to `20`, swap will be used when RAM becomes (roughly) 80 percent full. If you set it to `50`, swap will start being used when half your RAM is being used, and so on.

To change this value on the fly, you can execute the following command:

```
sudo sysctl vm.swappiness=30
```

This method doesn't set `swappiness` permanently, however. When you execute that command, `swappiness` will immediately be set to the new value and your server will act accordingly. Once you reboot, though, the `swappiness` value will revert back to the default. To make the change permanent, open the following file with your text editor:

```
/etc/sysctl.conf
```

A line in that file corresponding to `swappiness` will typically not be included by default, but you can add it manually. To do so, add a line such as the following to the end of the file and save it:

```
vm.swappiness = 30
```

Changing this value is one of many techniques within the realm of performance tuning. While the default value of `60` is probably fine for most, there may be a situation where you're running a performance-minded application and can't afford to have it swap any more than it absolutely has to. In such a situation, you would try different values for `swappiness` and use whichever one works best during your performance tests.

In the next section, we'll take a look at another important metric to keep an eye on: load average. The load average gives us an idea of how busy the CPU(s) might be, so we can better understand how to tell when our server is overwhelmed and we may need to take action.

## Understanding load average

Another very important topic to understand when monitoring performance is **load average**, which is a series of numbers that represents your server's trend in CPU utilization over a given time. You've probably already seen these series of numbers before, as there are several places in which the load average appears. If you run the `htop` utility, for example, the load average is shown on the screen. In addition, if you execute the `uptime` command, you can see the load average in the output of that command as well. You can also view your load average by viewing the text file that stores it in the first place:

```
cat /proc/loadavg
```

Personally, I habitually use the `uptime` command to view the load average. This command not only gives me the load average but also tells me how long the server has been running.

The load average is broken down into three sections, each representing 1 minute, 5 minutes, and 15 minutes respectively. A typical load average may look something like the following:

```
0.36, 0.29, 0.31
```

In this example, we have a load average of `0.36` in the 1-minute section, `0.29` in the five-minute section, and `0.31` in the fifteen-minute section. In particular, each number represents how many tasks were waiting for attention from the CPU for that given time period. Therefore, these numbers are really good. The server isn't that busy, since virtually no task is waiting for the CPU at any one moment (each number is less than 1). This is contrary to something such as overall CPU percentages, which you may have seen in task managers on other platforms. While viewing your CPU usage percentage can be useful, the problem with this is that your CPUs will constantly go from a high percentage of usage to a low percentage of usage, which you can see for yourself by just running `htop` for a while. When a task does some sort of processing, you might see your cores shoot up to 100 percent and then right back down to a lower number. That really doesn't tell you much, though. With load averages, you're seeing the trend of usage over three given time frames, which is more accurate in determining whether your server's CPUs are running efficiently or are choking on a workload they just can't handle.

The main question, though, is when you should be worried, which really depends on what kind of CPUs are installed on your server. Your server will have one or more CPUs, each with one or more cores. To Linux, each of these cores, whether they are physical or virtual, is the same thing (a CPU). In my case, the machine I took the earlier output from has a CPU with four cores. The more CPUs your server has, the more tasks it's able to handle at any given time, which also means it can handle a higher load average.

When a load average for a particular time period is equal to the number of CPUs on the system, that means your server is at capacity. It's handling a consistent number of tasks that are equal to the number of tasks it can handle. For example, if you have an 8-core CPU and the load average is 8 for a given time frame, then the CPU is 100% at its available capacity for that time frame. If your load average is consistently more than the number of cores you have available, that's when you'd probably want to look into the situation. It's fine for your server to be at capacity every now and then, but if it always is, that's a cause for alarm.

I'd hate to use a cliché example in order to fully illustrate this concept, but I can't resist, so here goes. A load average on a Linux server is equivalent to the check-out area at a supermarket. A supermarket will have several registers open, where customers can pay to finalize their purchases and move along. In my experience, you would have something like 20 check-out registers but only two cashiers working at any one time, but for this example, we'll assume each register has a cashier operating it.

Each cashier is only able to handle one customer at a time. If there are more customers waiting to check out then there are cashiers, the lines will start to back up and customers will get frustrated. In a situation where there are four cashiers and four customers being helped at a time, the cashiers would be at capacity, which is not really a big deal since no one is waiting. What can add to this problem is a customer that is paying by check and/or using a few dozen coupons, which makes the checkout process much longer (similar to a resource-intensive process). If there were four cashiers and six customers waiting, then there would be two more customers than the store is able to handle at the same time. This is essentially how load average works. Each cashier is a CPU, and each customer is a process that needs CPU time.

Just like the cashiers, each CPU can only handle one task at a time, with some tasks hogging the CPU longer than others. If there are exactly as many tasks as there are CPUs, there's no cause for concern. But if the lines start to back up, we may want to investigate what is taking so long. To take action, we may hire an additional cashier (add a new CPU) or ask a disgruntled customer to leave (kill a process).

Let's take a look at another example load average:

```
1.87, 1.53, 1.22
```

In this situation, we shouldn't be concerned, because our hypothetical server has four CPUs, and none of them have been at capacity within the 1-, 5-, or 15-minute time periods. Even though the load is consistently higher than 1, we have CPU resources to spare, so it's no big deal. If we had one of those awesome new Threadripper CPUs from AMD, which can have 32 cores (or maybe more by the time you're reading this), then those numbers would represent *extremely* low load. Going back to our supermarket comparison, the load average in the previous example would be equivalent to having four cashiers with an average of almost two customers being assisted during any 1 minute. If this server only had one CPU, we would probably want to figure out what's causing the line to begin to back up.

While having a low load average is usually a good thing, it can actually represent a really big problem depending on the context. When we deploy servers, we do so to get some sort of work done.

Whether that "work" is to host an application or run jobs to process data, our servers need to be doing some sort of work, otherwise, we're wasting money by having them. If the load average of your server drops to an abnormally low value, that might mean that a service that would normally be running all the time has failed and exited. For example, if you have a database server that constantly has a load within the 1.x range that suddenly drops to 0.x, that might mean that you either have legitimately less traffic or the database server service is no longer running. This is why it's always a good idea to develop baselines for your server, in order to gauge what is normal and what isn't. A baseline refers to resource usage, most of the time. If the resource usage is drastically higher or even lower than the baseline, that's a potential cause for concern either way.

Overall, load averages are something you'll become very familiar with as a Linux administrator if you haven't already. As a snapshot in time of how heavily utilized your server is, it will help you to understand when your server is running efficiently and when it's having trouble. If a server is having trouble keeping up with the workload you've given it, it may be time to consider increasing the number of cores (if you can) or scaling out the workload to additional servers. When troubleshooting utilization, planning for upgrades, or designing a cluster, the process always starts with understanding your server's load average so you can plan your infrastructure to run efficiently for its designated purpose.

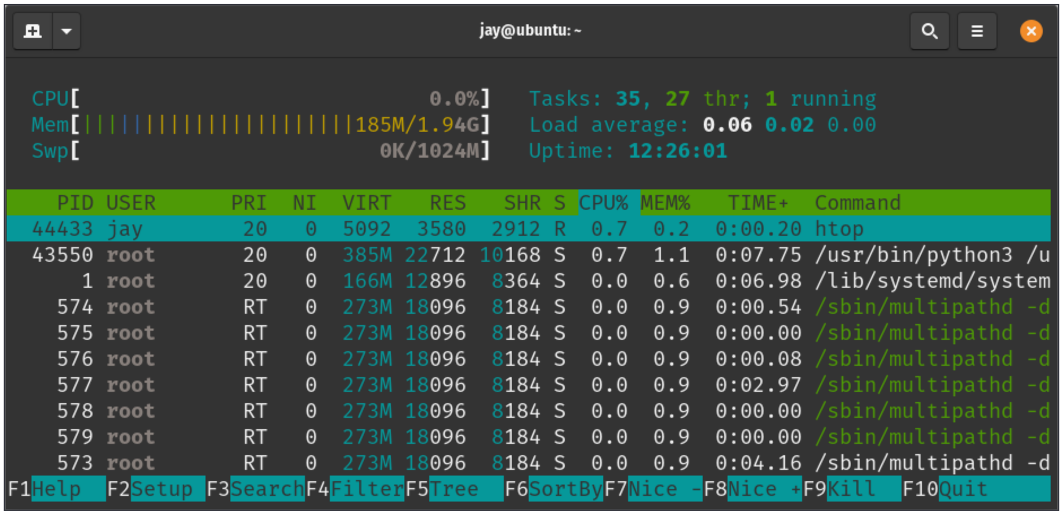
Now that we've gone over the important resources that we need to monitor to ensure our server remains healthy, let's take a look at a useful utility we can utilize that will make resource usage even easier to understand.

## Viewing resource usage with htop

When wanting to view the overall performance of your server, nothing beats `htop`. Although not typically installed by default, `htop` is one of those utilities that I recommend everyone installs as soon as possible, since it's indispensable when wanting to check on the resource utilization of your server. If you don't already have `htop` installed, all you need to do is install it with `apt`:

```
sudo apt install htop
```

When you run `htop` at your shell prompt, you will see the `htop` application in all its glory. In some cases, it may be beneficial to run `htop` as `root`, since doing so does give you additional options such as being able to kill processes, though this is not required:



```
  CPU[                          0.0%]   Tasks: 35, 27 thr; 1 running
  Mem[||||||||||||||||||||||185M/1.94G]   Load average: 0.06 0.02 0.00
  Swp[                        0K/1024M]   Uptime: 12:26:01

    PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  44433 jay        20   0  5092  3580  2912 R  0.7  0.2  0:00.20 htop
  43550 root       20   0  385M 22712 10168 S  0.7  1.1  0:07.75 /usr/bin/python3 /u
      1 root       20   0  166M 12896  8364 S  0.0  0.6  0:06.98 /lib/systemd/system
    574 root       RT   0  273M 18096  8184 S  0.0  0.9  0:00.54 /sbin/multipathd -d
    575 root       RT   0  273M 18096  8184 S  0.0  0.9  0:00.00 /sbin/multipathd -d
    576 root       RT   0  273M 18096  8184 S  0.0  0.9  0:00.08 /sbin/multipathd -d
    577 root       RT   0  273M 18096  8184 S  0.0  0.9  0:02.97 /sbin/multipathd -d
    578 root       RT   0  273M 18096  8184 S  0.0  0.9  0:00.00 /sbin/multipathd -d
    579 root       RT   0  273M 18096  8184 S  0.0  0.9  0:00.00 /sbin/multipathd -d
    573 root       RT   0  273M 18096  8184 S  0.0  0.9  0:04.16 /sbin/multipathd -d
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

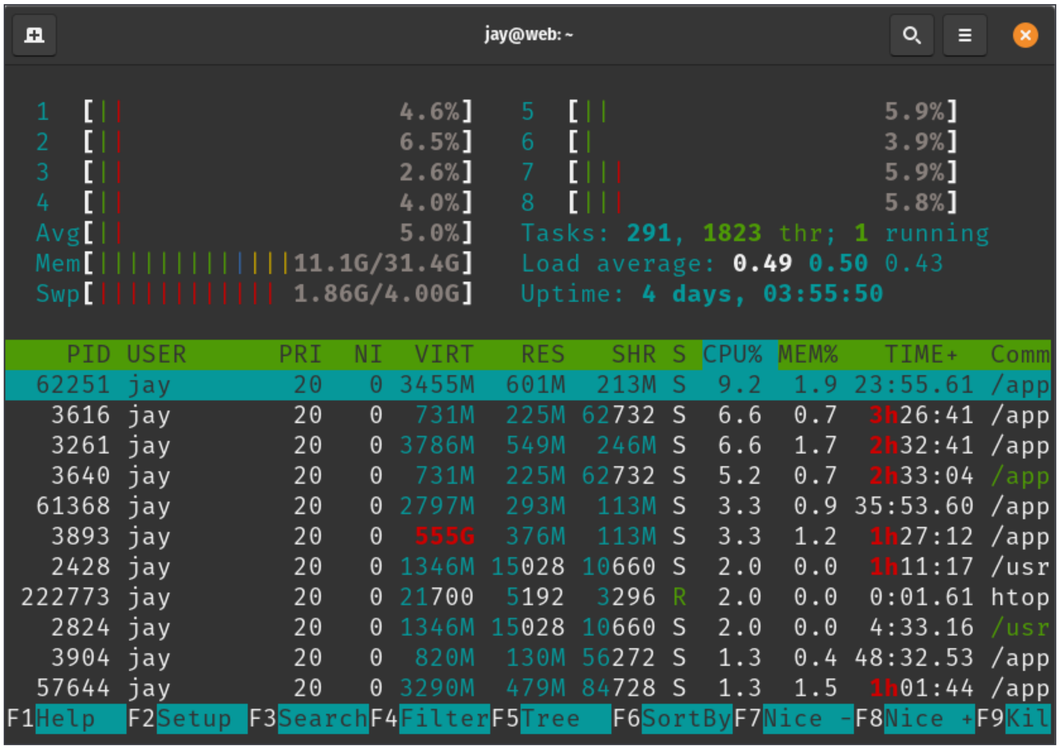
Figure 8.7: Running htop

At the top of the `htop` display, you'll see a progress meter for each of your cores (the server used for my screenshot only has one core), as well as a meter for memory as well as swap. In addition, the upper portion will also show your `Uptime`, `Load average`, and the number of `Tasks` you have running. The lower section of `htop`'s display will show you a list of processes running on your server, with fields showing you useful information such as how much memory or CPU is being consumed by each process, as well as the command being run, the user running it, and its **Process ID** (**PID**). We discussed PIDs in *Chapter 7*, *Controlling and Managing Processes*. To scroll through the list of processes, you can press *Page Up* or *Page Down* or use your arrow keys. In addition, `htop` features mouse support, so you are also able to click on columns at the top in order to sort the list of processes by that criteria. For example, if you click on `MEM%` or `CPU%`, the process list will be sorted by memory or CPU usage respectively. The contents of the display will be updated every 2 seconds.

The `htop` utility is also customizable. If you prefer a different color scheme, for example, you can press *F2* to enter **Setup mode**, navigate to **Colors** on the left, and then you can switch your color scheme to one of the six that are provided. Other options include the ability to add additional meters, add or remove columns, and more. One tweak I find especially helpful on multicore servers is the ability to add an average CPU bar. Normally, `htop` shows you a meter for each core on your server, but if you have more than one, you may be interested in the average as well. To do so, enter **Setup mode** again (*F2*), then with **Meters** highlighted, arrow to the right to highlight **CPU average** and then press *F5* to add it to the left column. There are other meters you can add as well, such as **Load average**, **Battery**, and more.

Depending on your environment, function keys may not work correctly in terminal programs such as `htop`, because those keys may be mapped to something else. For example, *F10* to quit `htop` may not work if *F10* is mapped to a function within your terminal emulator, and using a virtual machine solution such as VirtualBox may also prevent some of these keys from working normally.

Here's an example of `htop` configured with a meter for the CPU average.:



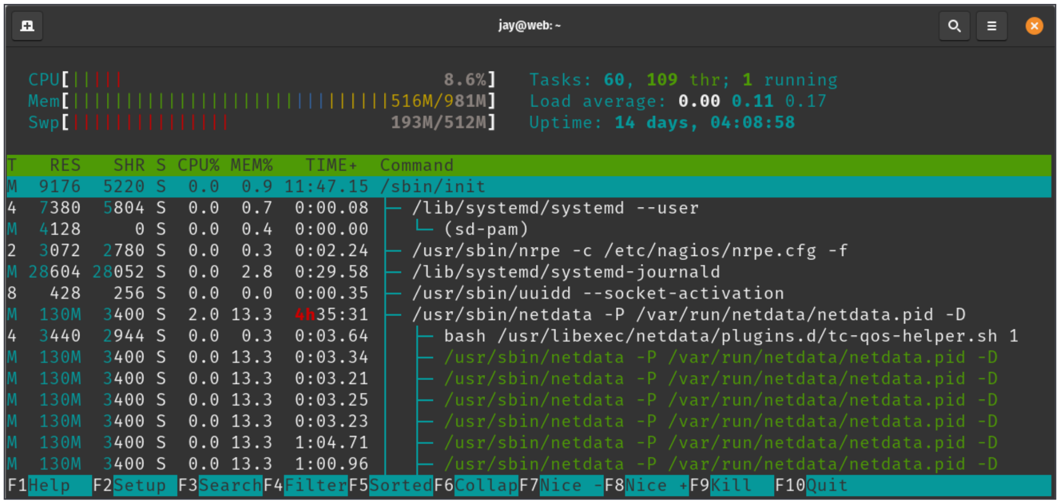
```
jay@web: ~

  1  [||                  4.6%]   5  [||                  5.9%]
  2  [||                  6.5%]   6  [|                   3.9%]
  3  [||                  2.6%]   7  [|||                 5.9%]
  4  [||                  4.0%]   8  [|||                 5.8%]
  Avg[||                  5.0%]   Tasks: 291, 1823 thr; 1 running
  Mem[|||||||||||||11.1G/31.4G]   Load average: 0.49 0.50 0.43
  Swp[|||||||||||| 1.86G/4.00G]   Uptime: 4 days, 03:55:50

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Comm
  62251 jay         20   0 3455M  601M  213M S  9.2  1.9 23:55.61 /app
   3616 jay         20   0  731M  225M 62732 S  6.6  0.7  3h26:41 /app
   3261 jay         20   0 3786M  549M  246M S  6.6  1.7  2h32:41 /app
   3640 jay         20   0  731M  225M 62732 S  5.2  0.7  2h33:04 /app
  61368 jay         20   0 2797M  293M  113M S  3.3  0.9 35:53.60 /app
   3893 jay         20   0  555G  376M  113M S  3.3  1.2  1h27:12 /app
   2428 jay         20   0 1346M 15028 10660 S  2.0  0.0  1h11:17 /usr
 222773 jay         20   0 21700  5192  3296 R  2.0  0.0  0:01.61 htop
   2824 jay         20   0 1346M 15028 10660 S  2.0  0.0  4:33.16 /usr
   3904 jay         20   0  820M  130M 56272 S  1.3  0.4 48:32.53 /app
  57644 jay         20   0 3290M  479M 84728 S  1.3  1.5  1h01:44 /app
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kil
```

Figure 8.8: htop with a meter for CPU average added

When you open `htop`, you will see a list of processes for every user on the system. When you have a situation where you don't already know which user/process is causing extreme load, this is ideal. However, a very useful trick (if you want to watch a specific user) is to press *u* on your keyboard, which will open up the **Show processes of:** menu. In this menu, you can highlight a specific user by highlighting it with the up or down arrow keys and then pressing *Enter* to only show processes for that user. This will greatly narrow down the list of processes.

Another useful view is the **tree view**, which allows you to see a list of processes organized by their parent/child relationship, rather than just a flat list. In practice, it's common for a process to be spawned by another process. In fact, all processes in Linux are spawned from at least one other process, and this view shows that relationship directly. In a situation where you are stopping a process only to have it immediately re-spawn, you would need to know what the parent of that process is in order to stop it from resurrecting itself. Pressing *F5* will switch `htop` to tree view mode, and pressing it again will disable the tree view.

With the tree view activated, `htop` will appear similar to the following:



```
jay@web: ~

  CPU[|||||                                  8.6%]   Tasks: 60, 109 thr; 1 running
  Mem[||||||||||||||||||||||||||||||||516M/981M]   Load average: 0.00 0.11 0.17
  Swp[|||||||||||||||                   193M/512M]   Uptime: 14 days, 04:08:58

T    RES   SHR S CPU% MEM%   TIME+  Command
M   9176  5220 S  0.0  0.9 11:47.15 /sbin/init
4   7380  5804 S  0.0  0.7  0:00.08 ├─ /lib/systemd/systemd --user
M   4128     0 S  0.0  0.4  0:00.00 │  └─ (sd-pam)
2   3072  2780 S  0.0  0.3  0:02.24 ├─ /usr/sbin/nrpe -c /etc/nagios/nrpe.cfg -f
M  28604 28052 S  0.0  2.8  0:29.58 ├─ /lib/systemd/systemd-journald
8    428   256 S  0.0  0.0  0:00.35 ├─ /usr/sbin/uuidd --socket-activation
M   130M  3400 S  2.0 13.3  4h35:31 ├─ /usr/sbin/netdata -P /var/run/netdata.pid -D
4   3440  2944 S  0.0  0.3  0:03.64 │  ├─ bash /usr/libexec/netdata/plugins.d/tc-qos-helper.sh 1
M   130M  3400 S  0.0 13.3  0:03.34 │  ├─ /usr/sbin/netdata -P /var/run/netdata/netdata.pid -D
M   130M  3400 S  0.0 13.3  0:03.21 │  ├─ /usr/sbin/netdata -P /var/run/netdata/netdata.pid -D
M   130M  3400 S  0.0 13.3  0:03.25 │  ├─ /usr/sbin/netdata -P /var/run/netdata/netdata.pid -D
M   130M  3400 S  0.0 13.3  0:03.23 │  ├─ /usr/sbin/netdata -P /var/run/netdata/netdata.pid -D
M   130M  3400 S  0.0 13.3  1:04.71 │  ├─ /usr/sbin/netdata -P /var/run/netdata/netdata.pid -D
M   130M  3400 S  0.0 13.3  1:00.96 │  ├─ /usr/sbin/netdata -P /var/run/netdata/netdata.pid -D
F1Help  F2Setup F3SearchF4FilterF5SortedF6CollapF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 8.9: htop with tree view activated

As I've mentioned, `htop` updates its stats every 2 seconds by default. Personally, I find this to be ideal, but if you want to change how fast it refreshes, you can call `htop` with the `-d` option and then apply a different number of seconds (entered in tenths of seconds) for it to refresh. For example, to run `htop` but have it update every 7 seconds, start `htop` with the following command:

```
htop -d 70
```

To kill a process with `htop`, use your up and down arrow keys to highlight the process you wish to kill and press *F9*. A new menu will appear, giving you a list of signals you are able to send to the process with `htop`. `SIGTERM`, as we discussed before, will attempt to gracefully terminate the process. `SIGKILL` will terminate it uncleanly. Once you highlight the signal you wish to send, you can send it by pressing *Enter* or cancel the process with *Esc*.

As you can see, `htop` can be incredibly useful and has (for the most part) replaced the legacy `top` command that was popular in the past. The `top` command is available by default in Ubuntu Server and is worth a look, if only as a comparison to `htop`. Like `htop`, the `top` command gives you a list of processes running on your server, as well as their resource usage. There are no pretty meters and there is less customization possible, but the `top` command serves the same purpose. In most cases, though, `htop` is probably your best bet going forward.

## Summary

In this chapter, we learned how to monitor our server's resource usage. We began with a look at the commands we can use to investigate disk usage, and we also learned how to monitor memory usage as well. We also discussed swap, including what it is, why you'd want to have it, as well as how to create a swap file manually should the need to do so come up. We then took a look at load average and closed out the chapter by checking out `htop`, which is my favorite utility for getting an overall look at resource usage on servers.

In *Chapter 9*, *Managing Storage Volumes*, we'll take a closer look at storage. In this chapter, we learned how to see how much is being used, but in the next we'll look at more advanced concepts surrounding storage, such as formatting volumes, adding additional volumes, and even LVM. See you there!

## Further reading

- Linux ate my RAM: `https://www.linuxatemyram.com/`

# 9
# Managing Storage Volumes

When it comes to storage on our servers, it seems as though we can never get enough. While hard disks are growing in capacity every year, and high capacity disks are cheaper than ever, our servers gobble up available space quickly. As administrators of servers, we always do our best to order servers with ample storage, but business needs evolve over time, and no matter how well we plan, a successful business will always need more. While managing your servers, you'll likely find yourself adding additional storage at some point. But managing storage is more than just adding new disks every time your current one gets full. Planning ahead is also important, and technologies such as **Logical Volume Manager** (**LVM**) will make your job much easier as long as you start using it as early as you possibly can.

In this chapter, I'll walk you through the concepts you'll need to know in order to manage storage and volumes on your server. This discussion will include:

- Adding additional storage volumes to the filesystem
- Formatting and partitioning storage devices
- Mounting and unmounting volumes
- Understanding the `/etc/fstab` file
- Backing up and restoring volumes
- Utilizing LVM

The first order of business for us in this chapter is to look into how to add additional storage to our server.

## Adding additional storage volumes

At some point or another, you'll reach a situation where you'll need to add additional storage to your server. On physical servers, we can add additional hard disks, and on virtual or cloud servers, we can add additional virtual disks. Either way, in order to take advantage of the extra storage we'll need to determine the name of the device, format it, and mount it. In the case of LVM (which we'll discuss later in this chapter), we'll have the opportunity to expand an existing volume, often without a server reboot being necessary. There's an overall process to follow when adding a new device, though. When adding additional storage to your system, you should ask yourself the following questions:

**How much storage do you need?** If you're adding a virtual disk, you can usually make it any size you want, as long as you have enough space remaining in the pool of your hypervisor.

**After you attached it, what device name did it receive?** When a new disk is attached to our server, it will be detected by the system and given a name. In most cases, the naming convention of `/dev/sda`, `/dev/sdb`, and so on will be used. In other cases (such as virtual disks), this will be different, such as `/dev/vda`, `/dev/xda`, and possibly others. The naming scheme usually ends with a letter, incrementing to the next letter with each additional disk.

**How do you want the storage device formatted?** At the time of writing, the ext4 filesystem is the most common. However, for different workloads, you may consider other options (such as XFS). When in doubt, use ext4, but definitely read up on the other options to see if they may benefit your use case. ZFS is another option that was introduced in version 16.04 of Ubuntu and is also present in 18.04 and now 20.04. ZFS won't be discussed in this book as it's currently considered to be in a state that is not ready for production use, but it will more than likely become a solid option once its implementation in Ubuntu matures. We'll discuss formatting in the next section, *Formatting and partitioning storage devices*.

It may be common knowledge to you by now, but the word filesystem is a term that can have multiple meanings on a Linux system depending on its context and may confuse newcomers. We use it primarily to refer to the entire file and directory structure (the Linux filesystem), but it's also used to refer to how a disk is formatted for use with the distribution (for example, the ext4 filesystem).

**Where do you want it mounted?** The new disk needs to be accessible to the system and possibly users, so you would want to mount (attach) it to a directory on your filesystem where your users or your application will be able to use it. In the case of LVM, which we also discuss in this chapter, you're probably going to want to attach it to an existing storage group. You can come up with your own directory for use with the new volume, but I'll discuss a few common locations later on in this chapter. We'll discuss mounting and unmounting in the *Mounting and unmounting volumes* section.

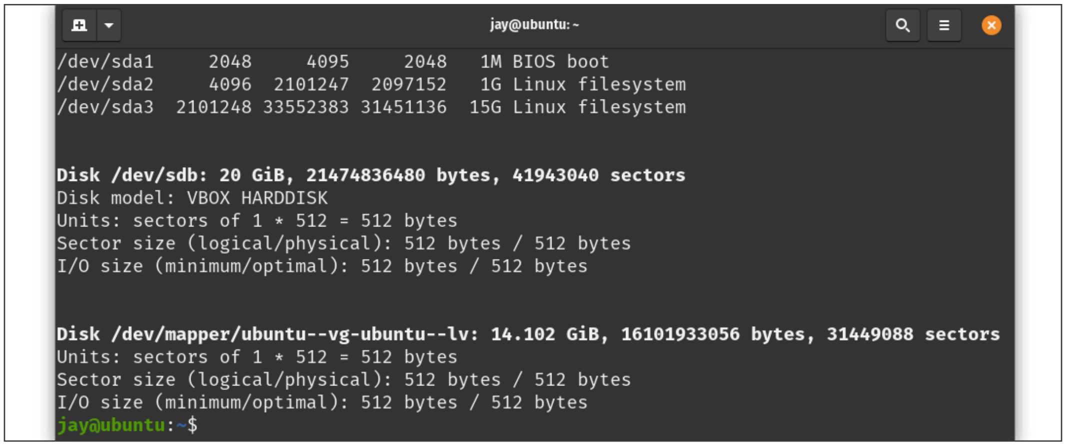
Let's consider the answers to the first two questions. With regards to how much space you should add, you would want to research the needs of your application or organization and find a reasonable amount. In the case of physical disks, you don't really get a choice beyond deciding which disk to purchase. In the case of virtual disks, you're able to be more frugal, as you can add a small disk to meet your needs (you can always add more later).

The main benefit of LVM with virtual disks is being able to grow a filesystem without a server reboot. For example, you can start with a 30 GB volume and then expand it in increments of 10 GB by adding additional 10 GB virtual disks. This method is certainly better than adding a 200 GB volume all at once when you're not completely sure all that space will ever be used. LVM can also be used on physical servers as well, but would most likely require a reboot anyway since you'd have to open the case and physically attach a hard drive. Some servers allow for hot-plugging, which gives you the ability to add/remove physical hard drives without powering off the server first, which is a great benefit to have.

Next, the device name can be found with the `fdisk -l` command. The `fdisk` command is normally used for creating and deleting partitions, but it will also allow us to determine which device name our new disk received. Using the `fdisk -l` command will give you the info, but you'll need to run it as `root` or with `sudo`:

```
sudo fdisk -l
```

Executing this command produces the following output:



```
/dev/sda1      2048     4095     2048   1M BIOS boot
/dev/sda2      4096  2101247  2097152   1G Linux filesystem
/dev/sda3   2101248 33552383 31451136  15G Linux filesystem


Disk /dev/sdb: 20 GiB, 21474836480 bytes, 41943040 sectors
Disk model: VBOX HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/mapper/ubuntu--vg-ubuntu--lv: 14.102 GiB, 16101933056 bytes, 31449088 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
jay@ubuntu:~$
```

Figure 9.1: Output of the fdisk -l command

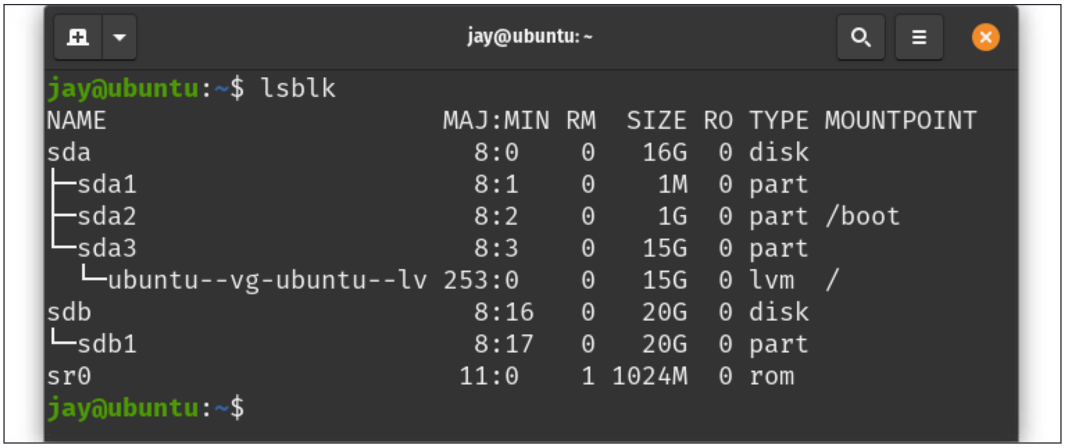
I always recommend running `fdisk -l` *before and after* attaching a new device. That way, it will be more obvious which device name represents the new device.

Another trick is to use the following command, with which the output will update automatically as you add the disk:

```
dmesg --follow
```

Just start the command, attach the disk, and watch the output. When done, press *Ctrl* + *c* on your keyboard to return back to the prompt.

You can also find the device name of your new disk with the `lsblk` command. One benefit of `lsblk` is that you don't need `root` privileges and the information it returns is simplified:



```
jay@ubuntu:~$ lsblk
NAME                        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda                           8:0    0    16G  0 disk
├─sda1                        8:1    0     1M  0 part
├─sda2                        8:2    0     1G  0 part /boot
└─sda3                        8:3    0    15G  0 part
  └─ubuntu--vg-ubuntu--lv 253:0    0    15G  0 lvm  /
sdb                           8:16   0    20G  0 disk
└─sdb1                        8:17   0    20G  0 part
sr0                          11:0    1  1024M  0 rom
jay@ubuntu:~$
```

Figure 9.2: Output of the lsblk command

On a typical server, the first disk (basically, the one that you installed Ubuntu Server on) will be given a device name of `/dev/sda` while additional disks will be given the next available name, such as `/dev/sdb`, `/dev/sdc`, and so on (depending on the type of hard disk you have). Nowadays, **Non-Volatile Memory Express** (**NVMe**) hard drives are becoming increasingly common, so you may see a device name such as `/dev/nvme0n1p1`. You'll also need to know the partition number. Device names for disks will also have numbers at the end, representing individual partitions. For example, the first partition of `/dev/sda` will be given `/dev/sda1`, while the second partition of `/dev/sdc` will be given `/dev/sdc2`. These numbers increment and are often easy to predict. As I mentioned before, your device naming convention may vary from server to server, especially if you're using a **Redundant Array of Independent Disks** (**RAID**) controller or a virtualization host such as VMware or XenServer. If you haven't created a partition on your new disk yet, you will not see any partition numbers at the end of their names.

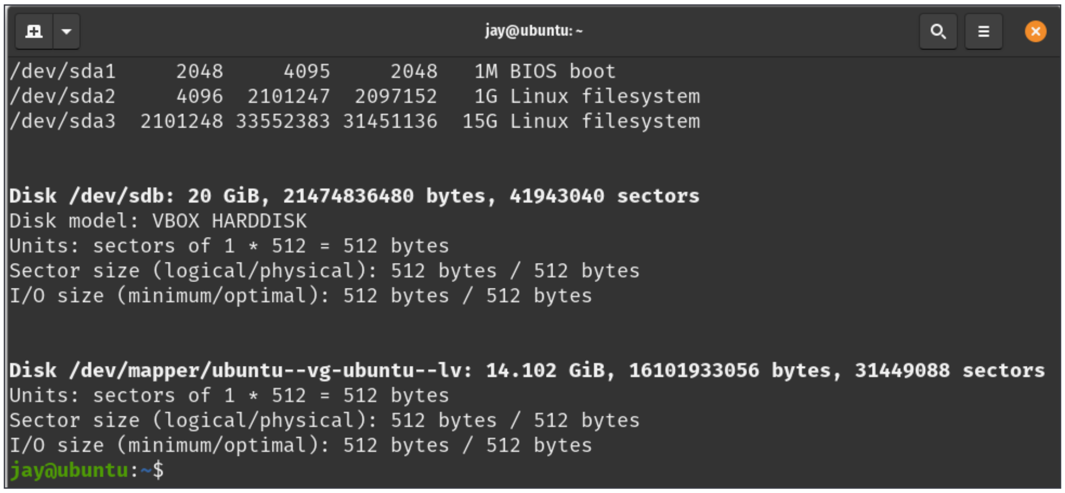
Now that you've added and named an additional storage volume, we can proceed through the process of setting it up. We need to decide where we're going to mount it, and what purpose it will serve. But before we can even mount the storage device in the first place, we need to create at least one partition on it, and then format it. We'll take care of both in the next section.

## Formatting and partitioning storage devices

Once you've installed a physical or virtual disk, you're well on your way to benefiting from additional storage. But in order to utilize a disk, it must first be formatted. In order to ensure we're formatting the correct disk, we need to find the name the device was given. As you already know from the previous section, there's a specific naming scheme that is used in Linux distributions to name disks. So you should already know the device name of the new disk. As explained earlier, you can use the `sudo fdisk -l` command to see details regarding the storage devices attached to your server:

```
sudo fdisk -l
```

This will produce an output that looks something like the following:



```
/dev/sda1     2048     4095     2048   1M BIOS boot
/dev/sda2     4096  2101247  2097152   1G Linux filesystem
/dev/sda3  2101248 33552383 31451136  15G Linux filesystem


Disk /dev/sdb: 20 GiB, 21474836480 bytes, 41943040 sectors
Disk model: VBOX HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/mapper/ubuntu--vg-ubuntu--lv: 14.102 GiB, 16101933056 bytes, 31449088 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
jay@ubuntu:~$
```

Figure 9.3: Using fdisk -l to view a list of storage devices on the server

In my case, the device `/dev/sdb` is brand-new—I just added it to the server. Since I'm using a VirtualBox virtual machine for the examples in this chapter, the new disk shows a model of `VBOX HARDDISK`. It doesn't have any partitions currently; notice how we see a few lines above it referring to a different hard disk and partitions, such as `/dev/sda3`. We don't have any lines like that in the description underneath for `/dev/sdb`. If we did have one or more partitions on that device, they would show up in the output.

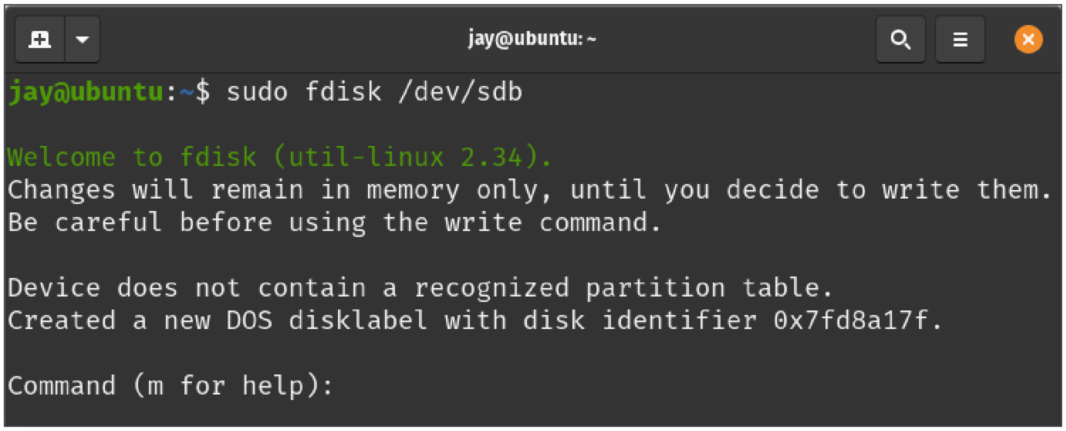
At this point, we know which storage device is new—there's no doubt that in the preceding example, it's `/dev/sdb`. We always need to make sure we don't attempt to format or repartition the wrong device, or we may lose data. In this case, we can see `/dev/sdb` has no partitions (and this volume wasn't present before I added it), so it's obvious which disk we'll want to be working with. Now we can create one or more partitions on it, to continue preparing it for use.

## Creating a partition

To create an actual partition on this device, we'll use the `fdisk` command with `sudo`, using the device's name as an option. In my case, I would execute the following to work with disk `/dev/sdb`:

```
sudo fdisk /dev/sdb
```

Note that I didn't include a partition number here, as `fdisk` works with the disk directly (and we also have yet to create any partitions). In this section, I'm assuming you have a disk that has yet to be partitioned or one you won't mind wiping. When executed correctly, `fdisk` will show you an introductory message and give you a prompt:



```
jay@ubuntu:~$ sudo fdisk /dev/sdb

Welcome to fdisk (util-linux 2.34).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0x7fd8a17f.

Command (m for help):
```

Figure 9.4: Main prompt of fdisk

At this point, you can press *m* on your keyboard for a menu of possible commands you can execute. In this example, I'll walk you through the commands required to set up a new disk for the first time.

I'm sure it goes without saying, but be aware of the destructive possibilities of `fdisk`. If you run `fdisk` against the wrong drive, irrecoverable data loss may result. It's common for an administrator to memorize utilities such as `fdisk` to the point where using it becomes muscle-memory. But always make sure that you take the time to ensure that you're running such commands against the appropriate disk.

Before we continue with creating a new partition, some discussion is required with regards to **Master Boot Record** (**MBR**) and **GUID Partition Table** (**GPT**) partition tables. When creating a partition table on a new disk, you'll have the option to set it up to use an MBR or GPT partition table. GPT is the newer standard, while MBR has been around for quite some time and is probably what you've been using if you've ever created partitions in the past.

You may see MBR referred to as DOS when referring to the older partition structure. As you may already know, **DOS** is short for **Disk Operating System**, but we're not referencing that operating system here in this chapter; we're referencing the partitioning structure that IBM came up with decades ago. For example, while using `fdisk`, it will refer to the MBR partition structure as DOS. In this chapter, we'll use MBR to refer to the older style whenever possible to avoid confusion.

With MBR partition tables, you have some limitations to consider. First, MBR only allows you to create up to four primary partitions. In addition, it also limits you to using somewhere around 2 TB of a disk. If the capacity of your disk is 2 TB or less, this won't be an issue. However, disks larger than 2 TB are becoming more and more common.

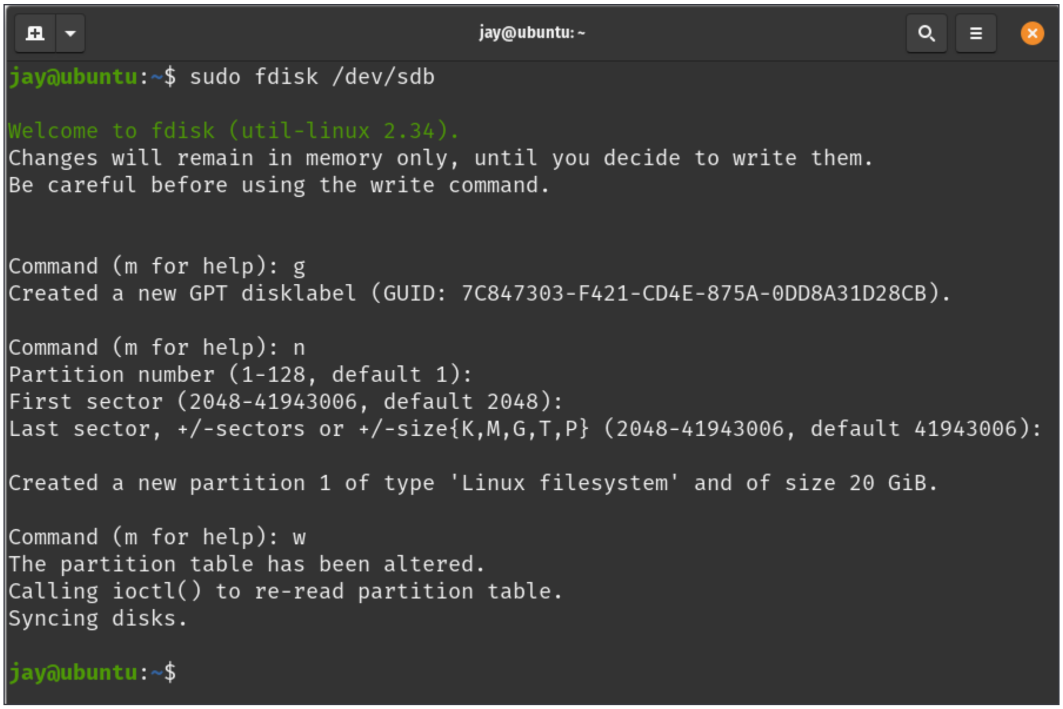
On the other hand, GPT doesn't have a 2 TB restriction, so if you have a very large disk, the decision between MBR and GPT has pretty much been made for you. In addition, GPT doesn't have a restriction of up to four primary partitions, as `fdisk` with a GPT partition table will allow you to create up to 128 of them. It's certainly no wonder GPT is fast becoming the new standard! It's only a matter of time before GPT becomes the default, so unless you have a good reason not to, I recommend using it if you have a choice.

When you first enter the `fdisk` prompt, you can press *o* to create an MBR-style partition layout, or you can press *g* to create the partition layout with the newer GPT style. As I've mentioned before, this is a potentially destructive process, so make sure you're using this utility against the correct drive! Make sure you press the associated key for your chosen partition style and then press *Enter*, and then we can proceed. Once you press *g* or *o*, you should see a confirmation that you have created a new partition table.

Continuing on, after you've made your choice and created either an MBR or GPT partition table, we're ready to proceed. Next, at the `fdisk` prompt, type *n* to tell `fdisk` that you would like to create a new partition. Then, you'll be asked if you would like to create a primary or extended partition (if you've opted for MBR). With MBR, you would want to choose primary for the first partition and then you can use extended for creating additional partitions. If you've opted for GPT, this prompt won't appear, as it will create your partition as primary no matter what.

The next prompt that will come up will ask you for the partition number, defaulting to the next available number. Press *Enter* to accept the default. Afterward, you'll be asked for the first sector for the partition to use (press *Enter* to accept the default of `2,048`) and then the next prompt will ask you for the last sector to use. If you press *Enter* to accept the default last sector, your partition will consist of all the free space that was remaining on the device. If you'd like to create multiple partitions, don't accept the default at the last sector prompt. Instead, you can clarify the size of your new partition by typing the `+` symbol followed by the number of mebibytes or gibibytes to use, and then `M` for mebibytes, or `G` for gibibytes. For example, you can enter `+20G` here to create a partition of 20 GiB. Note that there is no space after the **+** symbol, nor is there a space between **20** and **G**.

At this point, you'll be returned to the `fdisk` prompt. To save your changes and exit `fdisk`, press *w* and then *Enter*. Now if you run the `fdisk -l` command as `root`, you should see the new partition you created. Here is some example output from the `fdisk` command from one of my servers, to give you an idea of what the entire process looks like:



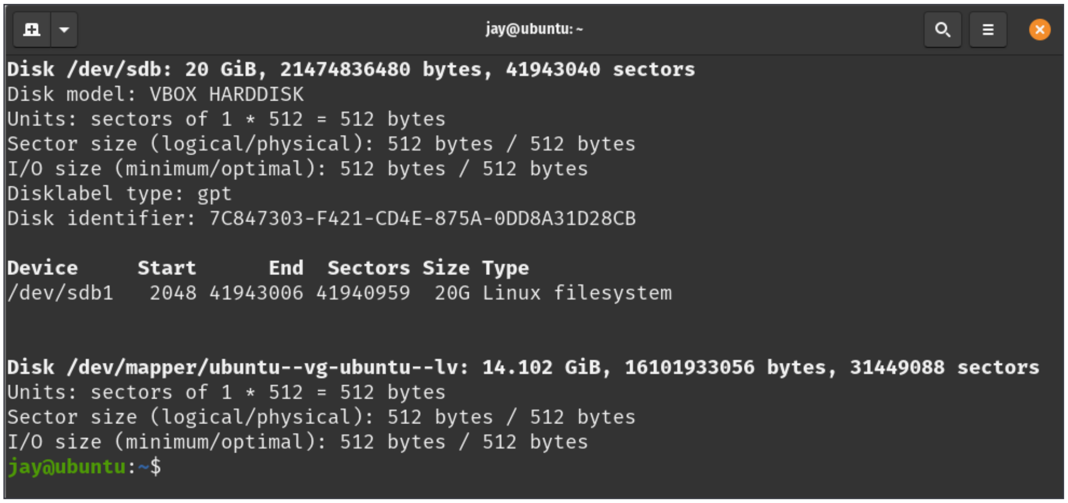
```
jay@ubuntu:~$ sudo fdisk /dev/sdb

Welcome to fdisk (util-linux 2.34).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help): g
Created a new GPT disklabel (GUID: 7C847303-F421-CD4E-875A-0DD8A31D28CB).

Command (m for help): n
Partition number (1-128, default 1):
First sector (2048-41943006, default 2048):
Last sector, +/-sectors or +/-size{K,M,G,T,P} (2048-41943006, default 41943006):

Created a new partition 1 of type 'Linux filesystem' and of size 20 GiB.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.

jay@ubuntu:~$
```

Figure 9.5: Example run of the fdisk command

If you've made a mistake or you want to redo your partition layout, you can do so by entering the `fdisk` prompt again and then pressing *g* to create a new GPT layout or *o* to create a new MBR layout. Then, continue through the steps again to partition your disk. Feel free to practice this a few times until you get the hang of the process.

## Formatting partitions

After you create your partition layout for your new disk and you're satisfied with it, you're ready to format it. Now that I've created a partition layout on the new disk, the output of `sudo fdisk -l` will be different:



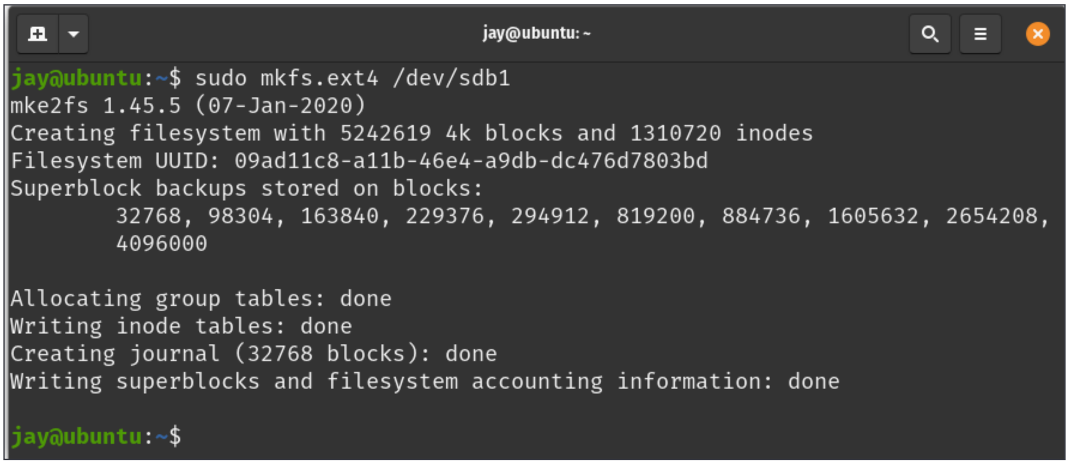
```
Disk /dev/sdb: 20 GiB, 21474836480 bytes, 41943040 sectors
Disk model: VBOX HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 7C847303-F421-CD4E-875A-0DD8A31D28CB

Device      Start      End  Sectors Size Type
/dev/sdb1    2048 41943006 41940959  20G Linux filesystem


Disk /dev/mapper/ubuntu--vg-ubuntu--lv: 14.102 GiB, 16101933056 bytes, 31449088 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
jay@ubuntu:~$
```

Figure 9.6: Another example of sudo fdisk -l, after creating a partition

Notice that now, we have the partition `/dev/sdb1` added, and visible in the output. Now, we can go ahead and format it. To do so, we take care of that with the `mkfs` command. This command is run with a specific syntax that entails typing `mkfs` along with a period (`.`), followed by the type of filesystem you would like to format the target as. The following example will format `/dev/sdb1` as ext4:

```
sudo mkfs.ext4 /dev/sdb1
```

Your output will look similar to mine in the following screenshot:



```
jay@ubuntu:~$ sudo mkfs.ext4 /dev/sdb1
mke2fs 1.45.5 (07-Jan-2020)
Creating filesystem with 5242619 4k blocks and 1310720 inodes
Filesystem UUID: 09ad11c8-a11b-46e4-a9db-dc476d7803bd
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
        4096000

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done

jay@ubuntu:~$
```

Figure 9.7: Formatting a volume using the ext4 filesystem

If you've opted for a filesystem type other than ext4, you can use that in place of ext4 when using `mkfs`. The following example creates an XFS filesystem instead:

```
sudo mkfs.xfs /dev/sdb1.
```

Some filesystems, such as XFS, are not supported by default and may need an additional package installed in order for it to be used. In the case of XFS, it requires the `xfsprogs` package to be installed.

So, now that we've created one or more partitions and formatted them, we're ready to mount the newly created partition(s) on our server. In the next section, I'll walk you through mounting and unmounting storage volumes.

## Mounting and unmounting volumes

Now that you've added a new storage volume to your server and have formatted it, you can mount the new device so that you can start using it. To do this, we use the `mount` command. This command allows you to attach a storage device (or even a network share) to a local directory on your server. Before mounting, the directory must be empty. The `mount` command, which we'll get to practice with an example very shortly, basically just requires you to designate a place (directory) for the device to be mounted to. But where should you mount the volume?

Normally, there are two directories created by default in your Ubuntu Server installation that exist for the purposes of mounting volumes: `/mnt` and `/media`. While there is no hard rule as far as where media needs to be mounted, these two directories exist as part of the **Filesystem Hierarchy Standard** (**FHS**) that was mentioned in *Chapter 4*, *Navigating and Essential Commands*. The purposes of the `/mnt` and `/media` directories are defined within this specification. The FHS defines `/mnt` as a mount point for a temporarily mounted filesystem, and `/media` as a mount point for removable media.

In plain English, this means that the intended purpose of `/mnt` is for storage volumes you generally keep mounted most of the time, such as additional hard drives, virtual hard disks, and network-attached storage. The FHS document uses the term *temporary* when describing `/mnt`, but I'm not sure why that's the case since this is generally where things are mounted that you generally expect to be around for a while. Perhaps temporary just refers to the fact that the system doesn't require this mount in order to boot, but who knows. In regard to `/media`, the FHS is basically indicating that removable media (flash drives, CD-ROM media, external hard drives, and so on) are intended to be mounted there.

However, it's important to point out that where the FHS indicates you should mount your extra volumes is only a suggestion. No one is going to force you to follow it, and the fate of the world isn't dependent on your choice. With the `mount` command, you can literally mount your extra storage anywhere that isn't already mounted or full of files. You could even create the directory `/kittens` and mount your disks there and you won't suffer any consequences other than a few chuckles from your colleagues.

Often, organizations will come up with their own scheme for where to mount extra disks. Although I personally follow the FHS designation, I've seen companies use a custom common mount directory such as `/store`. Whatever scheme you use is up to you; the only suggestion I can make is to be as consistent as you can from one server to another, if only for the sake of sanity.

The `mount` command generally needs to be run as `root`. While there is a way around that (you can allow normal users to mount volumes, but we won't get into that just yet), it's usually the case that only `root` can or should be mounting volumes. As I mentioned, you'll need a place to mount these volumes, so create a directory called `/mnt/vol1` with the following command:

```
sudo mkdir /mnt/vol1
```

When you've created a directory, like I have, or decided on an existing one, you can mount a volume with a command similar to the following:

```
sudo mount /dev/sdb1 /mnt/vol1
```

In that example, I'm mounting device `/dev/sdb1` to directory `/mnt/vol1`.

Of course, you'll need to adjust the command to reference the device you want to mount and where you want to mount it. As a reminder, if you don't remember which devices exist on your server, you can list them with `fdisk -l`.

Normally, the `mount` command wants you to issue the `-t` option with a given type. In my case, the `mount` command would've been the following had I used the `-t` option, considering my disk is formatted with ext4:

```
sudo mount /dev/sdb1 -t ext4 /mnt/vol1
```

A useful trick when mounting devices is to execute the `df -h` command before and after mounting.

While that command is generally used to check how much free space you have on various mounts, it does show you a list of mounted devices, so you can simply compare the results after mounting the device to confirm that it is present.

In that example, I used the `-t` option along with the type of filesystem I formatted the device with. In the first example, I didn't. This is because, in most cases, the `mount` command is able to determine which type of filesystem the device uses and adjust itself accordingly. Thus, most of the time, you won't need the `-t` option. In the past, you almost always needed it, but it's easier nowadays. The reason I bring this up is that if you ever see an error when trying to mount a filesystem that indicates an invalid filesystem type, you may have to specify this. Feel free to check the man pages for the `mount` command for more information regarding the different types of options you can use.

When you are finished using a volume, you can unmount it with the `umount` command (the missing *n* in the word *unmount* is intentional):

```
sudo umount /mnt/vol1
```

The `umount` command, which also needs to be run as `root` or with `sudo`, allows you to disconnect a storage device from your filesystem. In order for this command to be successful, the volume should not be in use. If it is, you may receive a device or resource busy error message. If you execute `df -h` after unmounting, you should see that the filesystem is missing from the output, and thus isn't mounted anymore.

The downside to manually mounting devices is that they will not automatically remount themselves the next time your server boots. In order to make the mount automatically come up when the system is started, you'll need to edit the `/etc/fstab` file, which I'll walk you through in the next section.

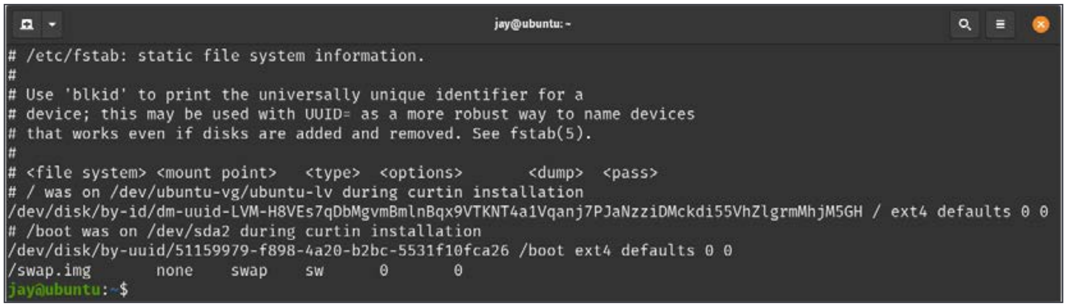
## Understanding the /etc/fstab file

The `/etc/fstab` file is a very critical file on your Linux system. You can edit this file to call out additional volumes you would like to automatically mount at boot time. However, the main purpose of this file is to also mount your main filesystem, so if you make a mistake while editing it, your server will not boot. Definitely be careful here.

## Analyzing the contents of /etc/fstab

When your system boots, it looks at the `/etc/fstab` file to determine where the root filesystem is. In addition, the location of your `swap` area is read from this file and mounted at boot time as well. Your system will also read any other mount points listed in this file, one per line, and mount them. Basically, just about any kind of storage you can think of can be added to this file and automatically mounted. Even network shares from Windows servers can be added here. It won't judge you (unless you make a typo).

As an example, here are the contents of `/etc/fstab` on one of my machines:



```
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/ubuntu-vg/ubuntu-lv during curtin installation
/dev/disk/by-id/dm-uuid-LVM-H8VEs7qDbMgvmBmlnBqx9VTKNT4a1Vqanj7PJaNzziDMckdi55VhZlgrmMhjM5GH / ext4 defaults 0 0
# /boot was on /dev/sda2 during curtin installation
/dev/disk/by-uuid/51159979-f898-4a20-b2bc-5531f10fca26 /boot ext4 defaults 0 0
/swap.img       none    swap    sw      0       0
jay@ubuntu:~$
```

Figure 9.8: Viewing the contents of the /etc/fstab file

When you install Ubuntu Server, the `/etc/fstab` file is created for you and populated with a line for each of the partitions the installer created during installation. On the server I used to grab the example `fstab` content, I have a single partition for the root filesystem, and you can also see where the swap file is mentioned.

Each partition is typically designated with a **Universally Unique Identifier** (**UUID**) instead of the `/dev/sdaX` naming convention you might be more accustomed to if you've worked with storage devices in the past. In my output, you can see UUID `dm-uuid-LVM-H8VEs7qDbMgv...`, which refers to my root filesystem, and you can also see that I have a `swap` file located at `/swap.img`.

The concept of a UUID has been around for a while, but there's nothing stopping you from replacing the UUID with the actual partition names (such as `/dev/sda1` or similar). If you were to do that, the server would still boot and you probably wouldn't notice a difference (assuming you didn't make a typo).

Nowadays, UUIDs are preferred over common device names due to the fact that the names of devices can change depending on where you place them physically (such as which particular **Serial Advanced Technology Attachment** (**SATA**) port a hard disk is plugged into, which of your USB ports an external drive is connected to, and so on) or how you order them (in the case of virtual disks).

Add to this the fact that removable media can be inserted or removed at any time and you have a situation where you don't really know what name each device is going to have at any one time. For example, your external hard drive may be named `/dev/sdb1` on your system now, but it may not be the next time you mount it if something else you connect claims the name of `/dev/sdb1`. This is one situation in which the concept of UUIDs comes in handy. A UUID of a device will not change if you reorder your disks (but it will change if you reformat the volume). As stated in *Figure 9.8*, you can easily list the UUIDs of your volumes with the `blkid` command:

```
blkid
```

The output will show you the UUID of each device attached to your system, and you can use this command any time you add new volumes to your server to list your UUIDs. This is also the first step in adding a new volume to your `/etc/fstab` file. While I did say that using UUIDs is not required, it's definitely recommended and can save you from trouble later on.

Each line of an `fstab` entry is broken down into several columns, each separated by spaces or tabs. There isn't a set number of spaces necessary to separate each column; in most cases, spaces are only used to line up each column to make them easier to read. However, at least one space is required.

In the first column of the example `fstab` file, we have the device identifier, which can be the UUID or label of each device that differentiates it from the others. (You can add a label to a device while formatting it with the `-L` argument with `mkfs` commands). In the second column, we have the location we want the device to be mounted to. In the case of the root filesystem, this is `/`, which (as you know) is the beginning of the Linux filesystem. The third entry in the screenshot (for `swap`) has a mount point of `none`, which means that a mount point is not necessary for this device. In the third column, we have the filesystem type, the first two being `ext4`, and the third having a type of `swap`.

In the fourth column, we have a list of options for each mount separated by a comma. In this case, we only have one option for each of the example lines. With the root filesystem, we have an option of `errors=remount-ro`, which tells the system to remount the filesystem as read-only if an error occurs. Such an issue is rare but will keep your system running in read-only mode as best it can if something goes wrong. The `swap` partition has a single option of `sw`. There are many other options that can be used here, so feel free to consult the man pages for a list. We will go over some of these options in this section.

The fifth and sixth columns refer to dump and pass respectively, which on my system are 0 and 0 for each line. The dump partition is almost always 0 and can be used with a backup utility to determine whether the filesystem should be backed up (0 for no, 1 for yes). In most cases, just leave this at 0 since this is rarely ever used by anything nowadays. The pass field refers to the order in which fsck will check the filesystems. The fsck utility scans hard disks for filesystem errors in the case of a system failure or a scheduled scan. The possible options for pass are 0, 1, or 2. With 0, the partition is never checked with fsck. If set to 1, the partition is checked first. Partitions with a pass of 2 are considered a second priority and checked last. As a general rule of thumb, consider using 1 for your main filesystem and 2 for all others. It's not uncommon for cloud server providers to use 0 for both fields. This may be because if a disk does undergo a routine check, it would take considerably longer to boot up. In a cloud environment, you can't always wait very long to get a server up and running.

Now that we understand all the columns of a typical fstab entry, we can work through adding another volume to the fstab file.

## Adding to the /etc/fstab file

To add another volume to the fstab file, we first need to know the UUID of the volume we would like to add (assuming it's a hard disk or virtual disk). Again, we do that with the blkid command:

```
blkid /dev/sdb1
```

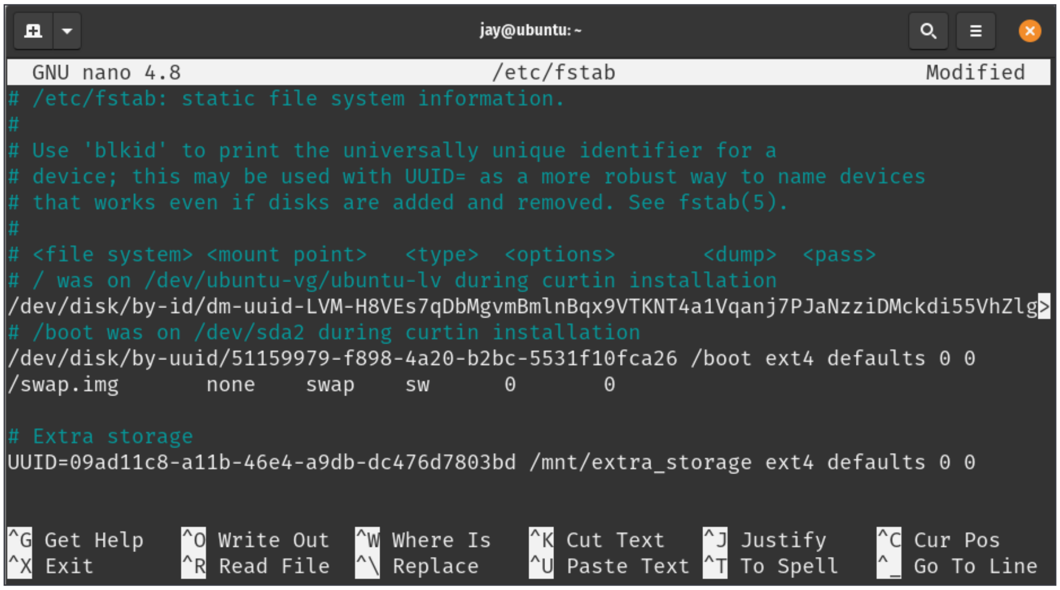
Notice that I used the device name of /dev/sdb1 as an argument. This is because I want to specifically fetch the UUID of the new device we added. The output of that command will give us the UUID of that device, so it can be added to the /etc/fstab file. Copy that down somewhere, as we'll need it shortly. Next, we need to know where we want to mount the volume. Go ahead and create the directory now, or use an existing directory if you wish. For example, you could create the directory /mnt/extra_storage for this purpose:

```
sudo mkdir /mnt/extra_storage
```

At this point, we should have all we need in order to add a new entry to fstab. To do so, we'll need to open the file in a text editor and then create a new line after all the others. If you don't have a preferred editor, you can use the nano editor:

```
sudo nano /etc/fstab
```

For example, the `/etc/fstab` file after adding an entry for `/dev/sdb` would look similar to the following:



```
  GNU nano 4.8                        /etc/fstab                        Modified
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/ubuntu-vg/ubuntu-lv during curtin installation
/dev/disk/by-id/dm-uuid-LVM-H8VEs7qDbMgvmBmlnBqx9VTKNT4a1Vqanj7PJaNzziDMckdi55VhZlg>
# /boot was on /dev/sda2 during curtin installation
/dev/disk/by-uuid/51159979-f898-4a20-b2bc-5531f10fca26 /boot ext4 defaults 0 0
/swap.img       none    swap    sw      0       0

# Extra storage
UUID=09ad11c8-a11b-46e4-a9db-dc476d7803bd /mnt/extra_storage ext4 defaults 0 0


^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Paste Text ^T To Spell   ^_ Go To Line
```

Figure 9.9: The /etc/fstab file after adding a new entry to it

In my example, I created a comment line with a little note about what the extra volume will be used for (`Extra storage`). It's always a good idea to leave comments, so other administrators will have a clue regarding the purpose of the extra storage. Then, I created a new line with the UUID of the volume, the mount-point for the volume, the filesystem type, `defaults` option, and a `dump`/`pass` of `0` and `0`.

The `defaults` option I've not mentioned before. By using `defaults` as your mount option in `fstab`, your mount will be given several useful options in one shot, without having to list them individually. Among the options included with `defaults` are the following, which are worth an explanation:

- `rw`: Device will be mounted read/write
- `exec`: Allow files within this volume to be executed as programs
- `auto`: Automatically mount the device at boot time
- `nouser`: Only `root` is able to mount the filesystem
- `async`: Output to the device should be asynchronous

Depending on your needs, the options included with defaults may or may not be ideal. Instead, you can call the options out individually, separated by commas, choosing only the ones you need. For example, with regards to `rw`, you may not want users to be allowed to change the content. In fact, I strongly recommend that you use `ro` (read-only) instead unless your users have a very strong use case for needing to make changes to files. I've actually learned this the hard way, where I've experienced an entire volume getting completely wiped out (and no one admitted to clearing the contents). This volume included some very important company data. From that point on, I mandated `ro` being used for everything, with a separate `rw` mount created, with only a select few (very responsible) people having access to it.

The `exec` option may also not be ideal. For example, if your volume is intended for storing files and backups, you may not want scripts to be run from that location. By using the inverse of `exec` (`noexec`), you can prevent scripts from running to create a situation where users are able to store files on the volume but not execute programs that are stored there.

Another option worth explanation is `auto`. The `auto` option basically tells your system to automatically mount that volume whenever the system boots or when you enter the following command:

```
sudo mount -a
```

When executed, `sudo mount -a` will mount any entry in your `/etc/fstab` file that has the `auto` option set. If you've used `defaults` as an option for the mount, those will be mounted as well since `defaults` implies `auto`. This way, you can mount all filesystems that are supposed to be mounted without rebooting your server (this command is safe to run whenever, as it will not disrupt anything that is already mounted).

The opposite of the `auto` option is `noauto`, which can be used instead. As you can probably guess by the name, an entry in `fstab` with the `noauto` option will not be automatically mounted and will not be mounted when you run `mount -a`. Instead, entries with this option will need to be mounted manually.

You may be wondering, then, what the point is of including an entry in `/etc/fstab` just to use `noauto`, which kind of seems to defeat the purpose. To explain this better, here is an example `fstab` entry with `noauto` being used:

```
UUID=e51bcc9e-45dd-45c7 /mnt/ext_disk  ext4  rw,noauto 0 0
```

Here, let's say that I have an external disk that I only mount when I'm performing a backup. I wouldn't want this device mounted automatically at boot time (I may not always have it connected to the server), so I use the `noauto` option. But since I do have an entry for it in `/etc/fstab`, I can easily mount it any time after I connect it with the following command:

```
sudo mount /mnt/ext_disk
```

Notice that I didn't have to include the device name or options; only the destination path for the mount. The `mount` command knows what device I'm referring to since I have an entry in the `/etc/fstab` file for a device to be mounted at `/mnt/ext_disk`. This saves me from having to type the device name and options each time I want to mount the device. So, in addition to mounting devices at boot time, the `/etc/fstab` file also becomes a convenient place to declare devices that may be used on an on-demand basis but aren't always attached.

One final option I would like to cover before we move on is `users`. When used with a mount in `/etc/fstab`, this allows regular users (users other than `root`) to mount and unmount the filesystem. This way, `root` or `sudo` will not be necessary at all for a mount used with this option. Use this with care, but it can be useful if you have a device with non-critical data you don't mind your users having full control over when mounting and unmounting.

While the concept of a text file controlling which devices are mounted on the system may seem odd at first, I think you'll appreciate being able to view a single file in order to find out everything that should be mounted and where it should be mounted. As long as administrators add all on-demand devices to this file, it can be a convenient place to get an overview of the filesystems that are in use on the server. As a bonus, you can also use the `mount` command (with no options) to have the system provide you with a list of everything that's mounted. Go ahead and try that, and I'll meet you in the next section.

## Backing up and restoring volumes

Since we're dealing with servers, the data that's being stored on our storage devices is no doubt going to be extremely important. While it's normal to have a few test servers for use as test subjects in a typical environment, our servers usually exist to carry out a very important task. I can tell you from first-hand experience, never put too much trust in storage devices. In fact, I recommend not trusting them at all. I consider all storage to be temporary, as hard drives can and do break. If your important data is only stored on one device, it's not safe. In this section, I'm going to discuss some very important topics around backups.

First, consider RAID volumes. We haven't discussed them in this chapter because while the technology can still be beneficial, it's not as popular as it once was. Don't get me wrong, there's still a place for RAID, but it's just not as popular as it used to be (especially with ZFS in the process of being approved by Canonical for production use). RAID allows you to join multiple disks in various configurations that can result in a lesser chance of losing data. For example, RAID level 1 ensures that two hard disks always have the same data. If one of the disks physically fails, then you haven't actually lost anything. When you replace a failed disk in RAID, it will rebuild the array with the new disk and then you'll again benefit from having some expandability. RAID 5 allows you to have multiple disks to benefit from more space, and RAID 6 is the same but it allows you to have two disks fail before you lose data, rather than just one. Generally, that's the difference between one level of RAID and another; how many disks are allowed to fail before it becomes a problem.

However, RAID suffers from some serious problems. The worst is that it's *not* a backup solution. It doesn't advertise itself to be that, but many administrators mistakenly assume that their data is safe when utilizing RAID. The truth is, the level of protection RAID offers you is minimal. If there's a lightning storm and a power surge gets past your surge protector and fries a hard disk, chances are the other one will fry too. Generally, the environmental factors that cause one hard disk to fail will likely cause other disks to fail too. Worse yet, if a criminal breaks into your server room, grabs your server, and runs away with it, then the crook got away with your server *and* all the disks in your RAID anyway, so there are various scenarios where it won't save you. RAID can definitely be good to have, but it's more of a convenience than anything else.

Backups that are actually good exist off of the server, somewhere else. The further away the backup is from the source server, the better. If you store your backups in a drawer outside of your server room, then that's certainly better than leaving an external backup disk connected all the time (which can also be susceptible to power surges just the same as an internal disk). But if a terrible storm takes out your entire building, then having the backup disk stored in the same physical location will work against you.

It may seem as though I'm being a bit overly dramatic here. But actually, I'm not. These situations can and do happen. Successful backups are resilient, and allow you to get your servers up and running quickly. Backups of your data are on a more important level than that, as some companies can go out of business if they lose their important files, which can include things like schematics that enable the company to be in business in the first place. As a system administrator, you'll need to develop a backup scheme that will account for as many scenarios as possible.

An effective backup routine will include several layers. Having an external disk is a useful backup, but why not have more than one, just in case one of them fails? Perhaps you can store one off-site, and swap the off-site and on-site backup disks weekly. In addition, perhaps you may use a command such as `rsync` to copy the files on your server to a remote server in another location periodically. You may even consider cloud backup solutions, which are another great addition.

In this section, I can't give you a backup scheme for your organization, because the layout of your backup system will depend on the needs of the organization, which are different from one company to the next. But what I can leave you with are some tips:

- Make sure to test your backups regularly. Simply having a backup isn't enough, they have to actually work! Try to restore data from a backup periodically to test the effectiveness.
- Have at least three layers in your backup scheme, with at least one being off-site. This can be a combination of external hard disks, network-attached storage, cloud storage, mirroring data to another server in another location, or whatever makes the most sense for your organization.
- Consider encryption. Although it's beyond the scope of this chapter, if your backups fall into the wrong hands, then protected data may leak and be readable by people you don't want to have the information.
- Check the policies of your organization, and ensure that your backup scheme is compliant. Not all companies have such a scheme, but if yours does, this is critical. Consider retention (how long backups must be kept for) and how frequently backups must be updated. If you don't have organizational policies, consult a lawyer to determine whether there are legal retention requirements for the industry of your organization.

Above all else, the point is to keep your data safe. So far in this book, we've looked at creating additional volumes, mounting them, and we even took a quick look at `rsync` earlier on. You've already learned some of the tools that can be made a part of a backup scheme, and you'll learn more methods before the book comes to a close. For now, keep these points in mind as you proceed through the book, and consider how each new skill you learn can be implemented as part of a backup scheme, if applicable.

**LVM** is one of my favorite technologies, giving us additional flexibility with our storage. In fact, let's take a look at that now.

## Utilizing LVM

The needs of your organization will change with time. While we as server administrators always do our best to configure resources with long-term growth in mind, budgets and changes in policy always seem to get in our way. LVM is something that I'm sure you'll come to appreciate. In fact, technologies such as LVM are those things that make Linux the champion when it comes to scalability and cloud deployments. With LVM, you are able to resize your filesystems online, without needing to reboot your server. LVM allows you to stop thinking of storage in terms of Linux, but instead to think of it in terms of scalability.

Take the following scenario for example. Say you have an application running on a virtualized production server, a server that's so important that downtime would cost your organization serious money. When the server was first set up, perhaps you gave the application's storage directory a 100 GB partition, thinking it would never need more than that. Now, with your business growing, it's not only using a lot of space—you're about to run out! What do you do? If the server was initially set up with LVM, you could add an additional storage volume, add it to your LVM pool, and grow your partition, all without rebooting your server! On the other hand, if you didn't use LVM, you're forced to find a maintenance window for your server and add more storage the old-fashioned way, which would include having it be inaccessible for a time.

With physical servers, you can install additional hard drives and keep them on standby without utilizing them to still gain the benefit of growing your filesystem online, even though your server isn't virtual. In addition, if your server supports hot-plugging, you can still add additional volumes without powering the server down.

It's for this reason that I must stress that you should always use LVM on storage volumes in virtual servers whenever possible. Let me repeat myself: you should *always* use LVM on storage volumes when you are setting up a virtual server! If you don't, this will eventually catch up with you when your available space starts to run out and you find yourself working over the weekend to add new disks. This will involve manually syncing data from one disk to another and then migrating your users to the new disk. This is not a fun experience, believe me. You might not think you'll be needing LVM right now, but you never know.

## Getting started with LVM

When setting up a new server via Ubuntu's installer, you're given the option to use LVM during installation. But it's much more important for your storage volumes to use LVM, and by those, I mean the volumes where your users and applications will store their data. LVM is a good choice for your Ubuntu Server's root filesystem if you'd like the root filesystem to also benefit from the features of LVM. In order to get started with LVM, there are a few concepts that we'll need to understand, specifically **volume groups**, **physical volumes**, and **logical volumes**.

A volume group is a namespace given to all the physical and logical volumes on your system. Basically, a volume group is the highest name that encompasses your entire implementation of an LVM setup. Think of it as a kind of container that is able to contain disks. An example of this might be a volume group named `vg-accounting`. This volume group would be used for a location for the accounting department to store their files. It will encompass the physical volumes and logical volumes that will be in use by these users. It's important to note that you aren't limited to just a single volume group; you can have several, each with their own disks and volumes.

A physical volume is a physical or virtual hard disk that is a member of a volume group. For example, the hypothetical `vg-accounting` volume group may consist of three 100 GB hard disks, and each would be considered a physical volume. Keep in mind that these disks are still referred to as physical volumes in the context of LVM, even when the disks are virtual. Basically, any block device that is owned by a volume group is a physical volume.

Finally, logical volumes are similar in concept to partitions. Logical volumes can take up a portion, or the whole, of a disk, but unlike standard partitions, they may also span multiple disks. For example, a logical volume can include three 100 GB disks and be configured such that you would receive a cumulative total of 300 GB. When mounted, users will be able to store files there just as they would a single partition on a standard disk. When the volume gets full, you can add an additional disk and then grow the partition to increase its size. Your users would see it as a single storage area, even though it may consist of multiple disks.

The volume group can be named anything you'd like, but I always give mine names that begin with `vg-` and end with a name detailing its purpose. As I mentioned, you can have multiple volume groups. Therefore, you can have `vg-accounting`, `vg-sales`, and `vg-techsupport` (and so on) all on the same server. Then, you assign physical volumes to each. For example, you can add a 500 GB disk to your server and assign it to `vg-sales`. From that point on, the `vg-sales` volume group owns that disk. You're able to split up your physical volumes any way that makes sense to you. Then, you can create logical volumes utilizing these physical volumes, which is what your users will use.

I think it's always best to work through an example when it comes to learning a new concept, so I'll walk you through such a scenario. In my case, I just created a local Ubuntu Server VM on my machine via VirtualBox and then I added four additional 20 GB disks after I installed the distribution. Virtualization is a good way to play around with learning LVM if you don't have a server available with multiple free physical disks.

To get started with LVM on a server that isn't already using it, you'll first need to have at least one additional (unused) volume, and install the required packages, which may or may not be present on your server. To find out if the required `lvm2` package is installed on your server, execute the following command:

```
apt search lvm2 |grep installed
```

If it's not present (the output of the previous command doesn't include `[installed,automatic]`), the following command will install the `lvm2` package and its dependencies:

```
sudo apt install lvm2
```

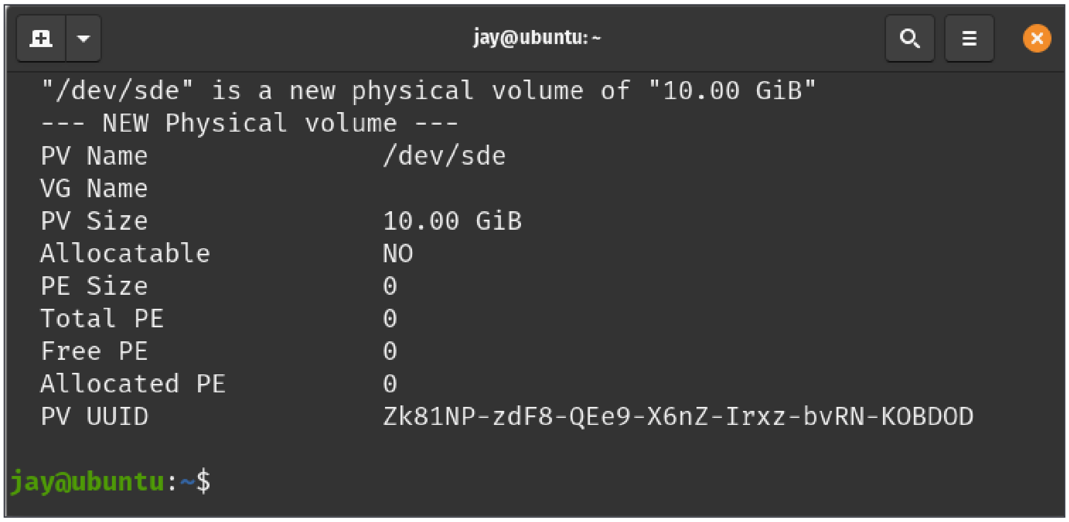
Next, we'll need to take an inventory of the disks we have available to work with. You can list them with the `fdisk -l` command as we've done several times now. In my case, I've added a few new volumes to my server, so now I have `/dev/sdb`, `/dev/sdc`, `/dev/sdd`, and `/dev/sde` to work with. The names of your disks will be different depending on your hardware or virtualization platform, so make sure to adjust all of the following commands accordingly.

To begin, we'll need to configure each disk to be used with LVM, by setting up each one as a physical volume. Note that we don't need to format a storage device, or even use `fdisk` to set it up before beginning the process of setting up LVM. Formatting actually comes later in this particular process. The `pvcreate` command is the first command we run to configure our disks for use with LVM. Therefore, we'll need to run the `pvcreate` command against all of the drives we wish to use for this purpose. For example, if I had four disks I wanted to use with LVM, I would run the following to set them up:

```
sudo pvcreate /dev/sdb
sudo pvcreate /dev/sdc
sudo pvcreate /dev/sdd
sudo pvcreate /dev/sde
```

And so on, for however many disks you plan on using.

To confirm that you have followed the steps correctly, you can use the `pvdisplay` command as `root` to display the physical volumes you have available on your server:



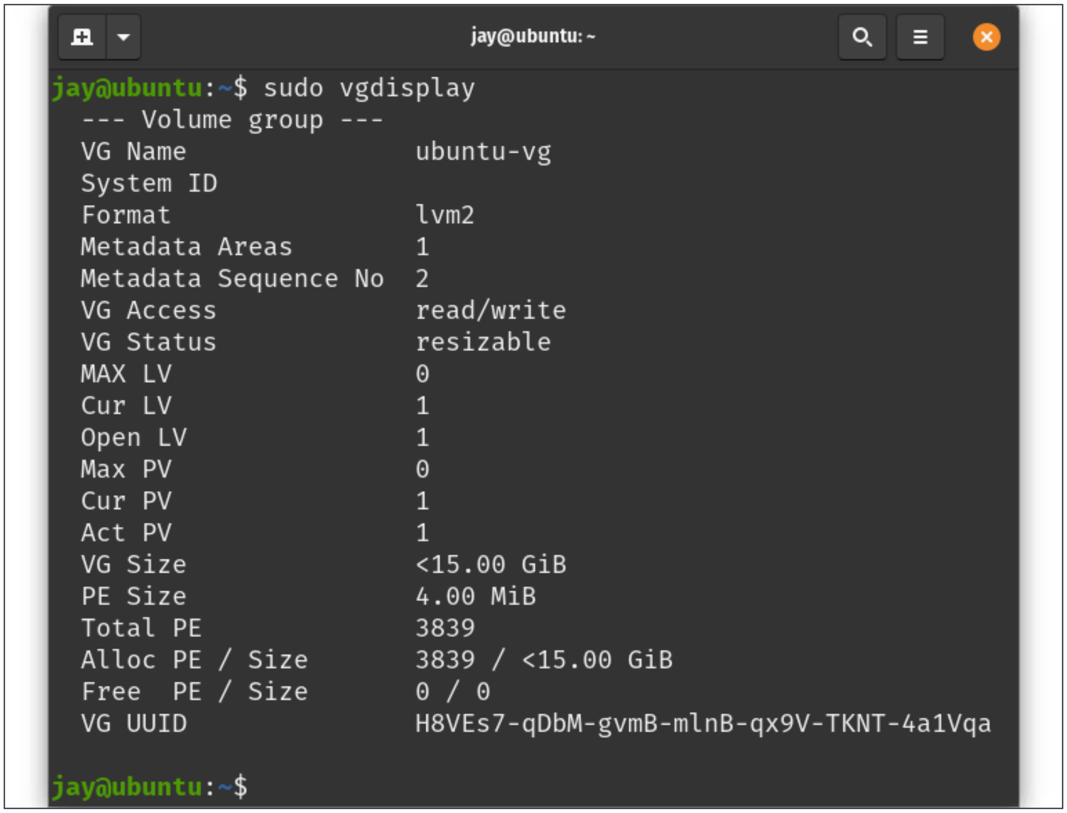
```
  "/dev/sde" is a new physical volume of "10.00 GiB"
  --- NEW Physical volume ---
  PV Name               /dev/sde
  VG Name
  PV Size               10.00 GiB
  Allocatable           NO
  PE Size               0
  Total PE              0
  Free PE               0
  Allocated PE          0
  PV UUID               Zk81NP-zdF8-QEe9-X6nZ-Irxz-bvRN-KOBDOD

jay@ubuntu:~$
```

Figure 9.10: Output of the pvdisplay command on a sample server

The screenshot shows only one volume, as it had to be formatted to fit this page. The `pvdisplay` command will show more output if you scroll up. Although we have some physical volumes to work with, none of them are assigned to a volume group. In fact, we haven't even created a volume group yet. We can now create our volume group with the `vgcreate` command, where we'll give our volume group a name and assign our first disk to it:

```
sudo vgcreate vg-test /dev/sdb
```

Here, I'm creating a volume group named `vg-test` and I'm assigning it one of the physical volumes I prepared earlier (`/dev/sdb`). Now that our volume group is created, we can use the `vgdisplay` command with `sudo` to view details about it, including the number of assigned disks (which should now be `1`):



```
jay@ubuntu:~$ sudo vgdisplay
  --- Volume group ---
  VG Name               ubuntu-vg
  System ID
  Format                lvm2
  Metadata Areas        1
  Metadata Sequence No  2
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                1
  Open LV               1
  Max PV                0
  Cur PV                1
  Act PV                1
  VG Size               <15.00 GiB
  PE Size               4.00 MiB
  Total PE              3839
  Alloc PE / Size       3839 / <15.00 GiB
  Free  PE / Size       0 / 0
  VG UUID               H8VEs7-qDbM-gvmB-mlnB-qx9V-TKNT-4a1Vqa

jay@ubuntu:~$
```

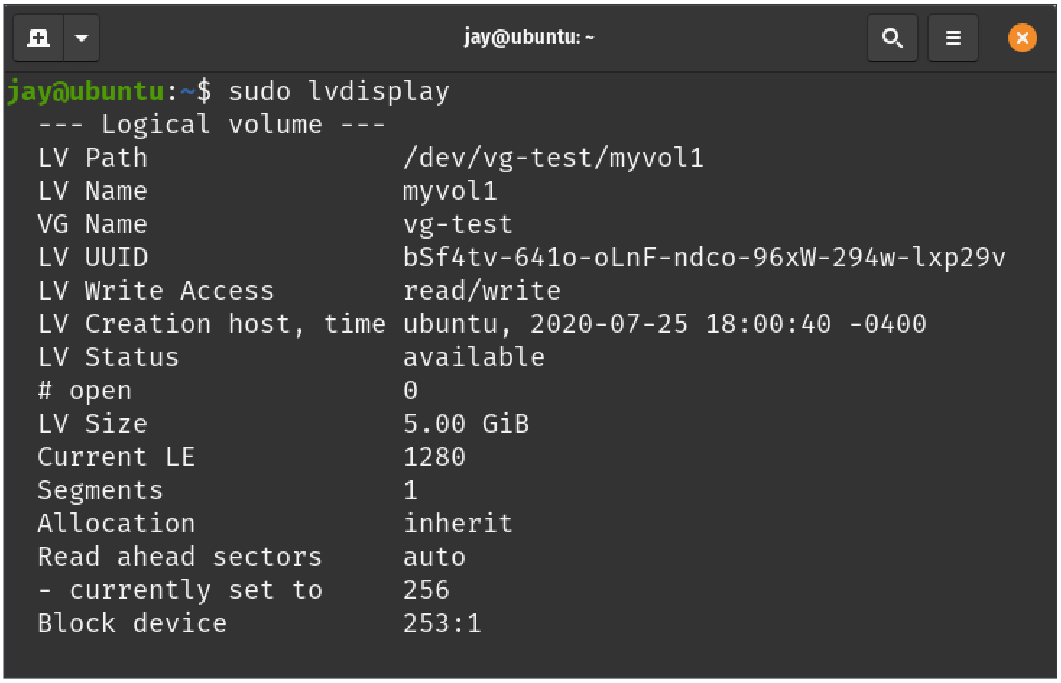
Figure 9.11: Output of the vgdisplay command on a sample server

At this point, if you created four virtual disks as I have, you have three more disks left that are not part of the volume group. Don't worry, we'll come back to them later. Let's forget about them for now as there are other concepts to work on at the moment.

All we need to do at this point is to create a logical volume and format it. Our volume group can contain all of, or a portion of, the disk we've assigned to it. With the following command, I'll create a logical volume of 5 GB from the virtual disk I added to the volume group:

```
sudo lvcreate -n myvol1 -L 5g vg-test
```

The command may look complicated, but it's not. In this example, I'm giving my logical volume the name `myvol1` with the `-n` option. Since I only want to give it 5 GB of space, I use the `-L` option and then `5g` to represent 5 GB. Finally, I give the name of the volume group that this logical volume will be assigned to. You can run `lvdisplay` with `sudo` to see information regarding this volume:



```
jay@ubuntu:~$ sudo lvdisplay
  --- Logical volume ---
  LV Path                /dev/vg-test/myvol1
  LV Name                myvol1
  VG Name                vg-test
  LV UUID                bSf4tv-641o-oLnF-ndco-96xW-294w-lxp29v
  LV Write Access        read/write
  LV Creation host, time ubuntu, 2020-07-25 18:00:40 -0400
  LV Status              available
  # open                 0
  LV Size                5.00 GiB
  Current LE             1280
  Segments               1
  Allocation             inherit
  Read ahead sectors     auto
  - currently set to     256
  Block device           253:1
```

Figure 9.12: Output of the lvdisplay command on a sample server

At this point, we should have everything we need as far as setting up LVM is concerned. But we still need to format a volume before we can use it, similar to a non-LVM disk.

## Formatting logical volumes

Next, we need to format our logical volume so that it can be used. However, as always, we need to know the name of the device so we know what it is we're formatting. With LVM this is easy. The `lvdisplay` command gave us this already; you can see it in the output (it's the third line down in *Figure 9.12*, under `LV Path`). Let's format it with the `ext4` filesystem:

```
sudo mkfs.ext4 /dev/vg-test/myvol1
```

And now this device can be mounted as any other hard disk. I'll mount mine at `/mnt/lvm/myvol1`, but you can use any directory name you wish:

```
sudo mount /dev/vg-test/myvol1 /mnt/lvm/myvol1
```

To check our work, execute `df -h` to ensure that our volume is mounted and shows the correct size. We now have an LVM configuration containing just a single disk, so this isn't very useful. The 5 GB I've given it will not likely last very long, but there is some remaining space we can use that we haven't utilized yet. With the following `lvextend` command, I can resize my logical volume to take up the remainder of the physical volume:

```
sudo lvextend -n /dev/vg-test/myvol1 -l +100%FREE
```

In this case, `+100%FREE` is the argument that clarified that we are wanting to use the entirety of the remaining space for the logical volume. If done correctly, you should see output similar to the following:

```
Logical volume vg-test/myvol1 successfully resized.
```

Now my logical volume is using the entire physical volume I assigned to it. Be careful, though, because if I had multiple physical volumes assigned, that command would've claimed all the space on those as well, giving the logical volume a size that is the total of all the space it has available, across all its disks. You may not always want to do this, but since I only had one physical volume anyway, I don't mind. Go ahead and check your free space again with the `df -h` command:

```
df -h
```

Unfortunately, it's not showing the extra space we've given the volume. The output of `df` is still showing the size the volume was before. That's because although we have a larger logical volume, and it has all the space assigned to it, we didn't actually resize the `ext4` filesystem that resides on this logical volume. To do that, we will use the `resize2fs` command:

```
sudo resize2fs /dev/mapper/vg--test-myvol1
```

The double-hyphen in the previous command is intentional, so make sure you're typing the command correctly.

If run correctly, you should see output similar to the following:

```
The filesystem on /dev/mapper/vg--test-myvol1 is now 5241856 (4k)
blocks long.
```

Now you should see the added space as usable when you execute `df -h`. The coolest part is that we resized an entire filesystem without having to restart the server. In this scenario, if our users have got to the point where they have utilized the majority of their free space, we will be able to give them more space without disrupting their work.

However, you may have additional physical volumes that have yet to be assigned to a volume group. In my example, I created four and have only used one in the LVM configuration so far. We can add additional physical volumes to our volume group with the `vgextend` command. In my case, I'll run this against the three remaining drives. If you have additional physical volumes, feel free to add yours with the same commands I use, but substitute my device names with yours:

```
sudo vgextend vg-test /dev/sdc
sudo vgextend vg-test /dev/sdd
sudo vgextend vg-test /dev/sde
```

You should see a confirmation similar to the following:

```
Volume group "vg-test" successfully extended
```

When you run `pvdisplay` now, you should see the additional physical volumes attached that weren't shown there before. Now that we have extra disks in our LVM configuration, we have some additional options. We could give all the extra space to our logical volume right away and extend it as we did before. However, I think it's better to withhold some of the space from our users. That way, if our users do use up all our available space again, we have an emergency reserve of space we could use at a pinch if we needed to while we figure out the long-term solution. In addition, LVM snapshots (which we will discuss soon) require you to have unallocated space in your LVM setup.

The following example command will add an additional 10 GB to the logical volume:

```
sudo lvextend -L+10g /dev/vg-test/myvol1
```

And finally, make the free space available to the filesystem:

```
sudo resize2fs /dev/vg-test/myvol1
```

With very large volumes, the resize may take some time to complete. If you don't see the additional space right away, you may see it gradually increase every few seconds until all the new space is completely allocated.

## Removing volumes with LVM

Finally, you may be curious about how to remove a logical volume or volume group. For these purposes, you would use the `lvremove` or `vgremove` commands. It goes without saying that these commands are destructive, but they are useful in situations where you want to delete a logical volume or volume group. To remove a logical volume, the following syntax will do the trick:

```
sudo lvremove vg-test/myvol1
```

Basically, all you're doing is giving the `lvremove` command the name of your volume group, a forward slash, and then the name of the logical volume within that group that you would like to remove. To remove the entire volume group, the following command and syntax should be fairly self-explanatory:

```
sudo vgremove vg-test
```

You can only remove a logical volume if it's not in use, and this may not be something you'll do very often, but if you ever do need to decommission an LVM component, then there are commands that will enable you to do so.

Hopefully, you're convinced by now how awesome LVM is. It allows you flexibility over your server's storage that other platforms can only dream of. The flexibility of LVM is one of the many reasons why Linux excels in the cloud market. These concepts can be difficult to grasp at first if you haven't worked with LVM before. But thanks to virtualization, playing around with LVM is easy. I recommend you practice creating, modifying, and destroying volume groups and logical volumes until you get the hang of it. If the concepts aren't clear now, they will be with practice.

In this section, you saw some ways in which LVM can benefit you; it allows you to take the storage of your server to the next level, even expanding it and growing it on demand. However, LVM also has additional tricks up its sleeve. It even allows you to create snapshots as well. We'll cover this useful ability next.

## Understanding LVM snapshots

**LVM snapshots** allow you to capture a logical volume at a certain point in time and preserve it. After you create a snapshot, you can mount it as you would any other logical volume and even revert your volume group to the snapshot in case something fails. In practice, this is useful if you want to test some potentially risky changes to files stored within a volume, but want the insurance that if something goes wrong, you can always undo your changes and go back to how things were. LVM snapshots allow you to do just that. LVM snapshots require you to have some unallocated space in your volume group.

However, LVM snapshots are definitely *not* a viable form of backup. For the most part, these snapshots are best when used as a temporary holding area when running tests or testing out experimental software before rolling out changes to production systems. During Ubuntu's installation process, you were offered the option to create an LVM configuration, so, therefore, you can use snapshots to test how security updates will affect your server if you used LVM for your root filesystem. If the new updates start to cause problems, you can always revert back. When you're done testing, you should merge or remove your snapshot.

So, why did I refer to LVM snapshots as a temporary solution and not a backup? First, similar to our discussion earlier, backups aren't secure if they are stored on the same server that's being backed up. It's always important to save backups off the server at least, preferably off-site. But what's worse is that if your snapshot starts to use up all available space in your volume group, it can get corrupted and stop working. Therefore, this is a feature you would use with caution, just as a means of testing something, and then revert back or delete the snapshot when you're done experimenting. Don't leave an LVM snapshot hanging around for too long.

When you create a snapshot with LVM, what happens is a new logical volume is created that is a clone of the original. Initially, no space is consumed by this snapshot. But as you run your server and manipulate files in your volume group, the original blocks are copied to the snapshot as you change them, to preserve the original logical volume. If you don't keep an eye on usage, you may lose data if you aren't careful and the logical volume will fill up.

To show this in an example, the following command will create a snapshot (called `mysnapshot`) of the `myvol1` logical volume:

```
sudo lvcreate -s -n mysnapshot -L 4g vg-test/myvol1
```

You should see the following output:

```
Logical volume "mysnapshot" created.
```

With that example, we're using the `lvcreate` command, with the `-s` option (snapshot) and the `-n` option (which allows us to name the snapshot), where we declare a name of `mysnapshot`. We're also using the `-L` option to designate a maximum size for the snapshot, which I set to 4 GB in this case. Finally, I give it the volume group and logical volume name, separated by a forward slash (`/`). From here, we can use the `lvs` command to monitor its size.

Since we're creating a new logical volume when we create a snapshot, we can mount it as we would a normal logical volume. This is extremely useful if we want to pull a single file without having to restore the entire thing.

But what about restoring the snapshot? One of the major benefits of snapshots is the ability to "roll back" to when the snapshot was taken. Essentially, this allows you to test changes to the server and then undo those changes. To roll back to a snapshot, we can do so with the `lvconvert` command:

```
sudo lvconvert --merge vg-test/mysnapshot
```

The output will look similar to the following:

```
Merging of volume mysnapshot started.
myvol1: Merged: 100.0%
```

It's important to note, however, that, unlike being able to resize a logical volume online, we cannot merge a snapshot while it is in use. If you do, the changes will take effect the next time it is mounted. Therefore, you can either unmount the logical volume before merging or unmount and remount after merging. Afterward, you'll see that the snapshot is removed the next time you run the `lvs` command.

> Since you cannot merge (roll back) a snapshot that is in use, if the snapshot is of the root filesystem, you'll have to reboot the server for the rollback to finalize.

If you'd like to make the snapshot permanent, which finalizes all of the changes you've made since the snapshot was first taken, we can use the `lvremove` command. For our example snapshot in this section, we can use the following command to make the snapshot permanent:

```
sudo lvremove vg-test/mysnapshot
```

As you can probably conclude based on the name of the command, `lvremove` deletes the snapshot. The act of deleting the snapshot is actually what makes its changes final, while the `lvconvert` command mentioned earlier rolls back to when the snapshot was taken.

LVM snapshots are definitely a useful feature, even if it's not supposed to be considered a backup solution. My favorite use case for these snapshots is to take a snapshot of the root filesystem before installing all available updates. After I reboot, and the updates take effect, I can either delete the snapshot (if everything seems to be fine) or revert back to the snapshot if the updates seem to be causing a problem. If nothing else, LVM snapshots are yet another trick you can use when and if the need for it comes up.

## Summary

Efficiently managing the storage of your servers will ensure that things continue to run smoothly, as a full filesystem can definitely cause your server to grind to a halt. Thankfully, Linux servers feature a very expansive toolset for managing your storage, some of which are a source of envy for other platforms. As Linux server administrators, we benefit from technologies such as LVM, and utilities such as `ncdu`, as well as many others. In this chapter, we explored these tools and how to manage our storage. We covered how to format, partition, mount, and unmount volumes, as well as managing the `fstab` file, LVM, monitoring disk usage, and more.

In the next episode of our Ubuntu Server saga, we'll work through connecting to networks. We'll configure our server's hostname, work through examples of connecting to other servers via OpenSSH, and take a look at IP addressing.

## Further reading

- Ubuntu LVM documentation: `https://wiki.ubuntu.com/Lvm`
- LVM video tutorial (from `LearnLinux.tv`): `https://learnlinux.link/lvm`

# 10
# Connecting to Networks

Linux networks are taking the IT industry by storm. Many organizations use Linux in their data centers, on both physical servers and in the cloud. Ubuntu Server is among the most popular choices for running mission-critical applications, but without a stable network to connect the individual components of your infrastructure together, even the most powerful server hardware will be ineffective.

So far in this book, we've worked with a single Ubuntu Server instance. Here, we begin a two-part look at networking in Linux. In this chapter, we'll discuss topics related to initial network connectivity and remote management. We'll continue learning additional networking topics in *Chapter 11*, *Setting Up Network Services*, where we'll work on building and configuring additional components that will enable your servers to communicate more effectively, which will result in a strong foundational network that will serve your needs for years to come.

In this episode of our Ubuntu adventure, we will cover:

- Setting the hostname
- Managing network interfaces
- Assigning static IP addresses
- Understanding Linux name resolution
- Getting started with OpenSSH
- Getting started with SSH key management
- Simplifying SSH connections with a config file

To get started in our exploration of networking, we should first give each of our Ubuntu servers their own identity; basically we should give them a name to help distinguish each from the others.

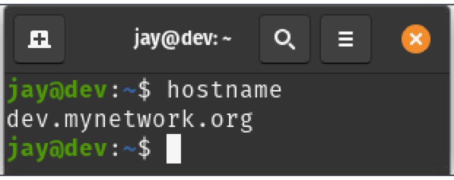
## Setting the hostname

During installation, you were asked to create a hostname for your server. Specifically, the field was labeled `Your server's name` during the initial setup process. At that time, our goal was to simply get an Ubuntu Server installation set up for working through the examples in this book. At this point, you may consider changing the hostname of your server. When we utilize OpenSSH to remotely manage our servers (as we'll do later on in this chapter) the hostname is shown on the command line. That can be very confusing if all servers have the same name. More importantly, the hostname of a server gives it an identity. When it comes to real production deployments of Ubuntu Server, each individual server should have its own designated purpose, and be named accordingly. Often, organizations will have their own naming scheme. Perhaps web servers in a company are named similar to `webserver-01`, or with a fully qualified domain name, such as `server1.mydomain.com`.

In this book, I won't assume any particular naming scheme, so when we do work through changing the hostname, feel free to adjust the name as you see fit. If you don't have a naming scheme (but would like to create one) feel free to get creative. I've seen quite a few variations, from naming servers after cartoon characters (who wouldn't want a server named `daffy-duck`?), to Greek gods or goddesses. Some companies choose to be a bit boring and come up with naming schemes consisting of a series of characters separated by hyphens, with codes representing which rack the server is in, as well as its purpose. You can create your own naming convention if you haven't already, and no matter what you come up with, I won't judge you.

As I mentioned, the hostname of your server is its identity. It identifies your server to the rest of the network. While a simple hostname, such as `ubuntu`, is fine if you have just one host, it would get confusing really quickly if you kept the default on every Ubuntu Server within your network. Giving each server a descriptive name helps you tell them apart from each other. But there's more to a server's name than its hostname, which we'll get into in *Chapter 11*, *Setting Up Network Services*, when we discuss DNS. But for now, we'll work through viewing and configuring the hostname, so you'll be ready to make your hostname official with a DNS assignment, when we come to it.

So, how do you view your hostname? One way is to simply look at your shell prompt; you've probably already noticed that your hostname is included there. While you can customize your shell prompt in many different ways, the default shows your current hostname. However, depending on what you've named your server, it may or may not show the entire name. Basically, the default prompt (known as a **PS1 prompt**, in case you were wondering) shows the hostname only until it reaches the first period. For example, if your hostname is `dev.mycompany.org`, your prompt will only show `dev`. To view the entire hostname, simply enter the `hostname` command:



```
jay@dev:~$ hostname
dev.mynetwork.org
jay@dev:~$
```

Figure 10.1: Output from the hostname command

Changing the hostname is fairly simple. To do this, we can use the `hostnamectl` command as `root` or with `sudo`. If, for example, I'd like to change my hostname from `dev.mynetwork.org` to `dev2.mynetwork.org`, I would execute the following command:

```
sudo hostnamectl set-hostname dev2.mynetwork.org
```

Simple enough, but what does that command actually do? Well, I'd love to give you a fancy outline, but all it really does is change the contents of a text file (specifically, `/etc/hostname`). To see this for yourself, feel free to use the `cat` command to view the contents of this file before and after making the change with `hostnamectl`:

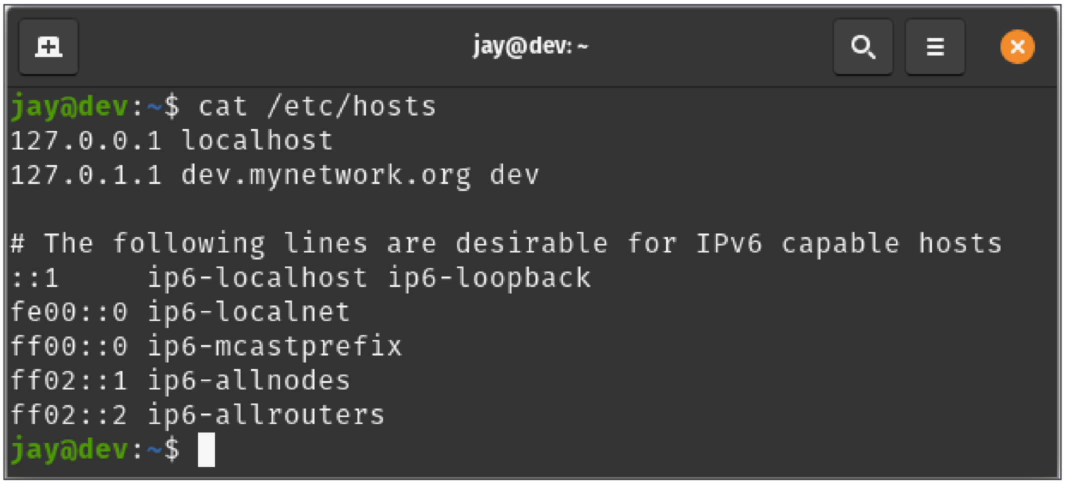
```
cat /etc/hostname
```

You'll see that this file contains only your hostname.

Once you change your hostname, you may start seeing an error message similar to the following after executing some commands:

```
unable to resolve host dev.mynetwork.org
```

This error means that the computer is no longer able to resolve your local hostname. This is due to the fact that the `/etc/hostname` file is not the only file where your hostname is located; it's also referenced in `/etc/hosts`. Unfortunately, the `hostnamectl` command doesn't update `/etc/hosts` for you, so you'll need to edit that file yourself to make the error go away. Here's what a `/etc/hosts` file looks like on an example server:



```
jay@dev:~$ cat /etc/hosts
127.0.0.1 localhost
127.0.1.1 dev.mynetwork.org dev

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
jay@dev:~$
```

Figure 10.2: Sample contents from a /etc/hosts file

The first two entries, in this example, refer to the local machine itself. Localhost addresses, also known as **loopback addresses**, allow the machine to essentially reach itself. If you were to use the `ping` command against the `127.0.0.1` address, the reply would come from the machine that executed the command, not from another host on the network. On the first line, we have the following:

```
127.0.0.1 localhost
```

If you were to use any networking command to attempt to communicate with the local server, such as pinging `127.0.0.1` or `localhost`, the `/etc/hosts` file on this line declares that this communication is directed toward the underlying server itself.

With the second line in the example screenshot, we have the following:

```
127.0.1.1 dev.mynetwork.org dev
```

Depending on your configuration, such as whether you are using a physical server, a virtualization platform, or a cloud server provider, that line may or may not be present. You can add that line if it's missing, but we'll talk about that more in a moment.

Essentially, that particular line identifies that the local server can also be reached at the IP address `127.0.1.1`, the fully qualified domain name of `dev.mynetwork.org`, as well as the simplified form of `dev`. A fully qualified domain name consists of the name of the server (`dev` in this case) as well as the domain name for the organization (`mynetwork.org` in this example). This enables you to ping your local server directly from that server by using the name `dev.mynetwork.org` or the simplified form of `dev`.

If you don't have a domain name to use with your servers, you can leave the fully qualified domain name out of the `/etc/hosts` file. So in our example, that line would look like the following with a domain:

```
127.0.1.1 dev
```

Going back to our example of changing a hostname on a server, I mentioned that you can use the `hostnamectl` command to do that, but that command doesn't update the `/etc/hosts` file for you, it only updates the `/etc/hostname` file. It's a best practice to also update the `/etc/hosts` file to match. You can avoid using the `hostnamectl` command altogether and manually edit the `/etc/hosts` and `/etc/hostname` files, which is actually my preferred method. If I have to manually edit a text file, regardless of whether or not I use the `hostnamectl` command, I figure that I may as well use a text editor for both.

The main takeaway, though, is to give your servers an identity that makes sense and matches the role that the server will fill within your network. At a typical organization, you'll have web servers, file servers, database servers, and more. A consistent and logical naming scheme will just make everything that much easier.

Now that we have learned how to give our servers an identity, we can learn how to manage network interfaces.

## Managing network interfaces

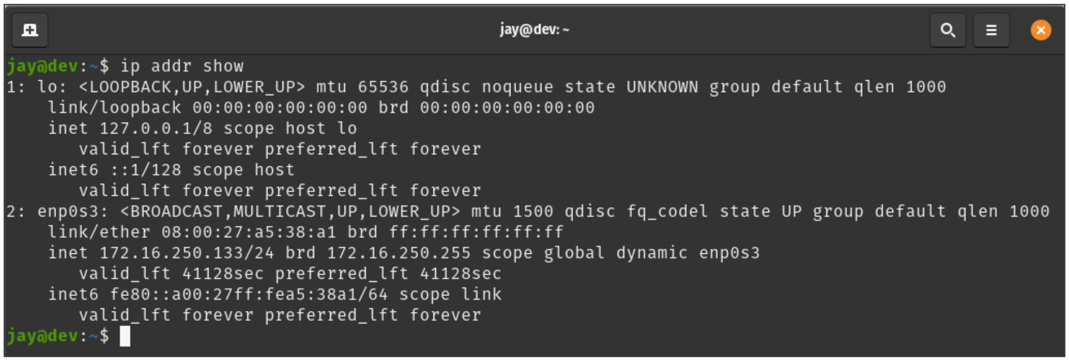
Networking is critical for server infrastructure. Without a network, servers cannot communicate with one another, and users will be unable to access them. In order for a server to connect to a network, it needs to have a network interface installed. Most servers will have a standard wired Ethernet adapter installed, allowing you to plug in a network cable to connect it to a switch. Assuming our server's hardware has been properly detected by Ubuntu, this is handled pretty much automatically. However, the automatic configuration is not always ideal. Perhaps we want to customize the IP address or settings related to the connection.

First, we need to understand how to view the current connection parameters that the network card of our server currently has in effect. That's the main goal in this section. We can do so using two basic commands: `ip` (which is recommended) and `ifconfig` (which was the previous method).

We can review information regarding our network interfaces and manage them with the `ip` command. For example, we can use `ip addr show` to view our currently assigned IP address:

```
ip addr show
```

This will produce an output similar to that shown in the following screenshot:



```
jay@dev:~$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:a5:38:a1 brd ff:ff:ff:ff:ff:ff
    inet 172.16.250.133/24 brd 172.16.250.255 scope global dynamic enp0s3
       valid_lft 41128sec preferred_lft 41128sec
    inet6 fe80::a00:27ff:fea5:38a1/64 scope link
       valid_lft forever preferred_lft forever
jay@dev:~$ 
```

Figure 10.3: Viewing IP information with the ip addr show command

Once you enter that command, you should see output that pertains to the network interfaces that you have available and their current status. Also, you can abbreviate the command all the way down to simply `ip a`. The output will be the same in either case. From the output, we can see several useful tidbits, such as the IP address for each device (if it has one), as well as its MAC address.

Using the `ip` command, we can also manage the state of an interface. We can bring a device down (prevent it from connecting to networks), and then back up again:

```
sudo ip link set enp0s3 down
sudo ip link set enp0s3 up
```

In that example, I'm simply toggling the state for interface `enp0s3`. First, I'm bringing it down, and then I'm bringing it back up again.

Bringing interfaces up and down is all well and good, but what's up with that naming convention? The convention used in Ubuntu 20.04 may seem a bit strange for those of you that have grown accustomed to the scheme used in earlier versions, which utilized network interface names such as `eth0`, `wlan0`, and so on. Since Ubuntu is based on Debian, it has adopted the new naming convention that was introduced starting with Debian 9.0.

The new naming convention has been put in place in order to make interface naming more predictable. While you may argue that names such as `eth0` may be easier to memorize than something like `enp0s3`, the change helps the name stay persistent between boots. When you add new network interfaces to a Linux system, there's always the possibility that other interface names may change as well. For example, if you have an older Linux installation on a server with a single network card (`eth0`) and you add a second (which is given the name `eth1`), your configuration may break if the names were to get switched during the next boot. Imagine for a moment that one interface is connected to the internet and another connected to a switch (basically, you have an internet gateway). If the interfaces came up in the wrong order, internet access would be disrupted for your entire office, due to the fact that the firewall rules you've written are being applied to the wrong interfaces. Definitely not a pleasant experience!

In the past, previous versions of Ubuntu (as well as Debian, and even CentOS), have opted to use `udev` to make the names stick in order to work around this issue. This is no longer necessary nowadays, but I figured I'd mention it here just in case you end up working on a server with an older installation. These older servers would achieve stickiness with interface names from configuration stored in the following file:

```
/etc/udev/rules.d/70-persistent-net-rules
```

This file existed on older versions of some popular Linux distributions (including Ubuntu), as a workaround to this problem. This file contains some information that identifies specific qualities of the network interface, so that with each boot, it will always come up with the same name. Therefore, the card you recognize as `eth0` will always be `eth0`. If you have an older version of Ubuntu Server in use, you should be able to see this file for yourself. Here's some sample output of this file on an older installation:

```
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*",
TTR{address}=="01:22:4e:a5:f2:ec", ATTR{dev_id}=="0x0",
ATTR{type}=="1", KERNEL=="eth*", NAME="eth0"
```

As you can see, it's using the MAC address of the card to identify it with `eth0`. But this becomes a small problem if I want to take an image of this machine and re-deploy it onto another server. This is a common practice—we administrators rarely start over from scratch if we don't have to, and if another server is similar enough to a new server's desired purpose, cloning it will be an option. However, when we restore the image onto another server, the `/etc/udev/rules.d/70-persistent-net-rules` file will come along for the ride. We'll more than likely find that the new server's first network interface will have a designation of `eth1`, even if we only have one interface.

This is because the file already designated a device as `eth0` (it's referencing a device that's not present in the system), so we would need to correct this file ourselves in order to reclaim `eth0`. We would do that by editing the rules file, deleting the line that contains the card that's not on the system, and then changing the device designation on the remaining line back to `eth0`.

The new naming scheme is effective as of `systemd v197` and later (in case you didn't already know from earlier topics in this book, `systemd` is the underlying framework utilized in Ubuntu for managing processes and various resources). For the most part, the new naming convention references the physical location of the network card on your system's bus. Therefore, the name it receives cannot change unless you were to actually remove the network card and place it in a different slot on the system's board, or change the position of the virtual network device in your hypervisor. If your machine does include a `/etc/udev/rules.d/70-persistent-net-rules` file, it will be honored and the old naming convention will be used instead. This is due to the fact that you may have upgraded to a newer version of your distribution (which features the new naming scheme, whereas your previous version didn't), so that your network devices will retain their names, thereby minimizing disruption when moving to a newer release.

As a quick overview of how the network names break down, `en` is for Ethernet, and `wl` is for wireless. Therefore, we know that the example interface I mentioned earlier (`enp0s3`) references a wired card. The `p` references which bus is being used, so `p0` refers to the system's first bus (the numbering starts at 0). Next, we have `s3`, which references PCI slot 3. Putting it together, `enp0s3` references a wired network interface card on the system's first bus, placed in PCI slot 3. The exact details of the new naming specification are beyond the scope of this chapter (and could even be a chapter of its own!), but hopefully this gives you a general idea of how the new naming convention breaks down. There's much more documentation online if you're interested in the nitty-gritty details (see the *Further reading* section). The important point here is that since the new naming scheme is based on where the card is physically located, it's much less likely to change abruptly. In fact, it can't change, as long as you don't physically switch the positions of your network cards inside the case.

Getting back to managing our interfaces, another command worth discussion is `ifconfig`.

The `ifconfig` command is part of the `net-tools` suite of utilities, which has been deprecated (for the most part). Its replacement is the `iproute2` suite of utilities, which includes the `ip` command we've already discussed. In summary, this basically means you should be using commands from the `iproute2` suite, instead of commands such as `ifconfig`. The problem, though, is that most administrators nowadays still use `ifconfig`, with no sign of it slowing down. In fact, the `net-tools` suite has been recommended for deprecation for years now, and just about every Linux distribution shipping today still has this suite installed by default. Those that don't have it installed offer it as an additional package that you can install. In the case of Ubuntu Server 20.04, the `net-tools` package is no longer installed by default, but it's still available if you want to manually install it. I don't recommend installing it though, since it's deprecated and shouldn't be used anymore.

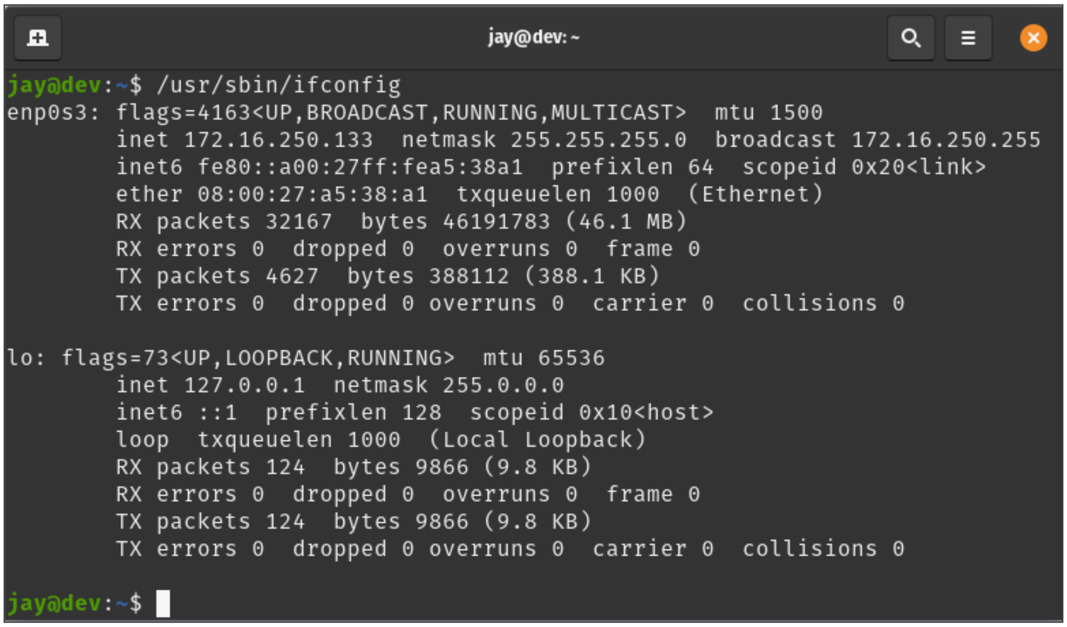
The reason commands such as `ifconfig` have a tendency to stick around so long after they've been deprecated usually comes down to the *change is hard* mentality, but quite a few scripts and programs out there are still using `ifconfig`, and therefore it's worth discussing here. Even if you immediately stop using `ifconfig`, and move to `ip` from now on, you'll still encounter this command on your travels, so you may as well know a few examples. Knowing the older commands will also help you if you find yourself on an older server.

First, when executed by itself with no options, `ifconfig` will print information regarding your interfaces like we did with `ip addr show` earlier. That seems pretty simple.

If you are unable to use `ifconfig` to view interface information using a normal user, try using the fully qualified command (include the full path):

```
/usr/sbin/ifconfig
```

The `/usr/sbin` directory may or may not be in your `$PATH` (a set of directories your shell looks within for commands), so if your system doesn't recognize `ifconfig`, using the fully qualified command should produce the desired output, as follows:



```
jay@dev:~$ /usr/sbin/ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.16.250.133  netmask 255.255.255.0  broadcast 172.16.250.255
        inet6 fe80::a00:27ff:fea5:38a1  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:a5:38:a1  txqueuelen 1000  (Ethernet)
        RX packets 32167  bytes 46191783 (46.1 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4627  bytes 388112 (388.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 124  bytes 9866 (9.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 124  bytes 9866 (9.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

jay@dev:~$ 
```

Figure 10.4: Viewing interface information with the ifconfig command

Secondly, just like with the `ip` commands we practiced earlier, we can also bring an interface down or up with `ifconfig` as well:

```
sudo ifconfig enp0s3 down
sudo ifconfig enp0s3 up
```

There are, of course, other options and variations of `ip` and `ifconfig`, so feel free to look up the man pages for either if you want more information. For the purposes of this section, the main thing is to remember how to view your current IP assignments, as well as how to bring an interface up or down.

As useful as our network interfaces can be, they're useless without an IP address assigned to them. While a network will often use DHCP to take care of this, in the next section we'll take a look at how to assign a static IP address.

## Assigning static IP addresses

With servers, it's very important that your IP addresses remain fixed and do not change for any reason. If an IP address does change (such as a dynamic lease with no reservation), your users will experience an outage, services will fail, or entire sites may become unavailable. When you install Ubuntu Server, it will grab a dynamically assigned lease from your DHCP server, but after you configure the server the way you want it, it's important to set a permanent IP address right away before it's considered production-ready. One exception to this rule is an Ubuntu-based VPS. Cloud providers that bill you for these servers will have an automatic system in place to declare an IP address for your new VPS, and will already have it configured to remain in place. But in the case of virtual or physical servers you manage yourself, you'll start off with a dynamic address, unless you've already configured a static IP address during installation.

In most cases, you'll have an IP address scheme in place at your office or organization, which will outline a range of IP addresses that are available for use with static assignments. If you don't have such a scheme, it's important to create one, so you will have less work to do later when you bring more servers online. We'll talk about setting up a DHCP server and IP address scheme in *Chapter 11*, *Setting Up Network Services*, but for now, I'll give you a few quick tips. Your DHCP server will typically have a range of IP addresses that will be automatically assigned to any host that requests an assignment. When setting up a static IP on a server, you'll want to make sure that the IP address that you choose is outside of the range that your DHCP server assigns so you don't end up with a duplicate IP on your network. For example, if your DHCP server assigns IPs ranging from `10.10.10.100` through `10.10.10.150`, you'll want to use an IP address *not* included within that range for your servers.

There are two ways of assigning a fixed address to a network host, including your servers. The first is by using a static IP assignment, as I've already mentioned. With that method, you'll arbitrarily grab an IP address that's not being used by anything, and then configure your Ubuntu Server to use that address. In that case, your server is never requesting an IP address from your network's DHCP server. It simply uses whatever you assign it. This is the method I'll be going over in this section.

The other way of assigning a fixed address to a server is by using a static lease. This is also known as a **DHCP reservation**, but I prefer to use the former term. With this method, you configure your DHCP server to assign a specific IP address to specific hosts. In other words, your server will request an IP address from your local DHCP server, and your DHCP server is instructed to give a specific address to your server each time it asks for one. This is the method I prefer, because it makes your DHCP server the single source of truth for the IP addresses that are assigned on your network. I'll go over it in more detail in *Chapter 11*, *Setting Up Network Services*.

However, you don't always have a choice. As a Linux administrator, you may or may not be in charge of the DHCP server. It's often the case at many organizations that administrators that manage servers are not the same individuals that also manage the network. If you don't have authority over the design of the network, you'd use an IP address that would be provided to you by your network administrator, and then you'll proceed to configure your Ubuntu Server to use it by applying the parameters they give you.

Within the last several years, the method that we utilize to customize the IP address of our servers has changed from the way it was handled in the past. Since Ubuntu 17.10, which was released back in 2017, configuring your IP address settings is now done via Netplan. In the past, we would configure networking via NetworkManager, but that's not installed by default anymore in Ubuntu. With Netplan, configuration files for your network interfaces now reside in the `/etc/netplan` directory, in YAML format. The YAML format itself is beyond the scope of this book, but the syntax is very easy to follow so you don't really need to thoroughly understand this format in order to configure your network interfaces. If you list the contents of the `/etc/netplan` directory, you should see at least one file there, often named `00-installer-config.yaml` or `50-cloud-init.yaml`. It's possible the file could be saved with a different name, so check the contents of the `/etc/netplan` directory to see what the file is named on your end. On one of my servers, I see the following contents in the `/etc/netplan/00-installer-config.yaml` file:

```
# This is the network config written by 'subiquity'
network:
  ethernets:
    enp0s3:
      dhcp4: true
  version: 2
```

We can already glean some obvious information from this default file. First, the comment at the beginning mentions `subiquity`, which is the official name of the Ubuntu Server installer, which is used when you install the distribution from boot media created from the ISO file.

More importantly, we can see that this particular server is configured to utilize DHCP in order to grab an IP address, which we can tell from the following line:

```
      dhcp4: true
```

We can also tell that this configuration file pertains to interface `enp0s3`. Putting it all together, this file is telling us that interface `enp0s3` is configured to automatically obtain an IP address via DHCP. If we'd like to convert this configuration to a static IP address instead, we should first back up the file:

```
sudo cp /etc/netplan/00-installer-config.yaml /etc/netplan/00-installer-config.yaml.bak
```

This way, if we make a mistake, we can easily restore the original file by renaming the backup file to the original name. This is a good practice to get into, regardless of the file we're editing. Being able to restore a previous configuration is a best practice for just about any change we might be making. The first change we need to make is to remove the following line:

```
      dhcp4: true
```

Essentially, to set up a static IP we will replace that line with the details specific to our configuration. Here's an example of the file, configured for a static IP address:

```
# This is the network config written by 'subiquity'
network:
  ethernets:
    enp0s3:
      addresses: [192.168.100.50/24]
      gateway4: 192.168.100.1
      nameservers:
        addresses: [192.168.100.1, 192.168.100.2]
  version: 2
```

In the example, I've bolded four lines, which were added in place of the `dhcp4: true` line. First, we set the actual IP address:

```
      addresses: [192.168.100.50/24]
```

There, I've used an example IP address of `192.168.100.50/24`. On your end, you would make sure that the IP address you choose is within the scope of the network you'd like your server to be a part of. As I mentioned earlier, the IP address you choose should *not* be within the DHCP scope that automatically assigned IP addresses are chosen from. The preceding IP address would be fine if the example scope of the DHCP server on your network ranged from `192.168.100.100` to `192.168.100.150`. The IP chosen here of `192.168.100.50` is outside of that, so we don't have to worry about another device being assigned that address.

We also include `/24` to declare that the IP address is part of a 24-bit **subnet**, which is fairly standard unless your network administrator has set up a larger scope. A `/24` network is the same as a Class C network, if that's more familiar to you. This also takes care of the subnet mask, which we don't need here since `/24` implies the same subnet mask as `255.255.255.0` (if you're more familiar with the classful style, which shows the subnet mask in the same format as an IP address). We will discuss subnets in *Chapter 11*, *Setting Up Network Services*, however, a full walk-through of subnetting and the TCP/IP protocol is beyond the scope of this book, so we won't worry too much about that value for now.

Moving on, we also set up the gateway:

```
      gateway4: 192.168.100.1
```

When it comes to networking, the gateway refers to the "next hop," which is basically the IP address that outbound connections are routed through. This value will need to match the actual default gateway address on your network, which is quite often the same as the IP address with the last portion being `.1`. If in doubt, you can check the IP address assignment of another device on the same network you're joining the Ubuntu server to, which would be the same.

The last section allows us to configure the DNS server that our server will use to look up external domain names:

```
      nameservers:
        addresses: [192.168.100.1, 192.168.100.2]
```

The example configuration is just that, an example—all the values must match whatever is appropriate for your network. Often, the DNS server IP address will be the same as the gateway address, but that's not always true. Sometimes a network administrator will have a custom IP scheme for DNS servers. I also added a secondary DNS server in the example, `192.168.100.2`, but you can remove the second IP address if that's not necessary.

Once you've ensured that the values in the file are appropriate, we'll need to apply and test these changes:

- If you are using a virtual machine, you may want to make the changes from the virtual machine console.
- If you're updating a physical machine, you may want to have a display and keyboard attached.
- Although we discuss OpenSSH later in this chapter, if you're already aware of how to connect to a server via OpenSSH, you probably won't want to change network configuration over OpenSSH, since as soon as you activate these changes your connection will drop.

Just take your time and double-check everything to ensure you didn't mistype anything, so you won't find yourself with a server that cannot connect.

To actually make these changes take effect, you can run the following command:

```
sudo netplan apply
```

When you run the previous command, it will let you know if there are any errors in the file or apply the changes if not. The new IP address will take effect immediately.

In the case of utilizing remote connections such as OpenSSH while configuring networking, you can work around the issue of being disconnected and having networking not restart properly by using `tmux`, a popular terminal multiplexer. A full run-through of `tmux` is beyond the scope of this book, but it is helpful to us in this scenario because it keeps commands running in the background, even if our connection to the server gets dropped. To use it, first install the package:

```
sudo apt install tmux
```

Then, activate `tmux` by simply typing `tmux` in your shell prompt.

From this point on, `tmux` is now responsible for your session. If you run a command within `tmux`, it will continue to run, regardless of whether or not you're attached to it. To see this in action, first enter `tmux` and then execute the `top` command. While `top` is running, disconnect from `tmux`. To do that, press *Ctrl* + *b* on your keyboard, release, and then press *d*. You'll exit `tmux`, but if you enter the `tmux a` command to reattach your session, you'll see that `top` was still running even though you disconnected. Following this same logic, you can activate `tmux` prior to executing the `sudo netplan apply` command. Most likely, you'll still get dropped from your shell, since the process of activating network changes brings the network interface down and then back up again, but with `tmux` the command will complete in the background. You can then reconnect to the server and run `tmux a` to rejoin your `tmux` session.

The `tmux` utility is extremely powerful, and when harnessed can really enhance your workflow when using the Linux shell. Although a complete tutorial is outside the scope of this book, I highly recommend looking into using it. For a full walkthrough, check out the book *Getting Started with tmux* by Victor Quinn, J.D. at `https://www.packtpub.com/hardware-and-creative/getting-started-tmux` or check out some tutorial videos on YouTube, such as my own series on the subject at `learnlinux.tv`.

With networking restarted, you should be able to immediately reconnect to the server and see that the new IP assignment has taken place by executing `ip a`. If, for some reason, you cannot reconnect to the server, you may have made a mistake while editing the configuration file for Netplan. Double-check that file for any errors. But as long as you've followed along and typed in the proper values for your interface and network, you should be up and running with a static IP assignment.

Now, we have an actual network—we've named our server(s) and configured our network interfaces. We should also understand how name resolution works in Ubuntu, which is the process in which servers are able to find other servers by their name.

## Understanding Linux name resolution

In *Chapter 11*, *Setting Up Network Services*, we'll have a discussion on setting up a DNS server for local name resolution for your network. But before we get to that, it's also important to understand how Linux resolves names in the first place. Most of you are probably aware of the concept of a **Domain Name System** (**DNS**), which matches human-understandable domain names to IP addresses. This makes browsing your network (as well as the internet) much easier. However, a DNS isn't always the first thing that your Linux server will use when resolving names.

For more information on the order in which Ubuntu Server checks resources to resolve names, feel free to take a look at the `/etc/nsswitch.conf` file. There's a line in this file that begins with the word `hosts`. Here is the output of the relevant line from the file on my server:

```
hosts:          files dns
```

In this case, the server is configured to first check local files, and then the DNS if the request isn't found. This is the default order, and I see little reason to make any changes here (but you certainly can). Specifically, the file the server will check is `/etc/hosts`. If it doesn't find what it needs there, it will move on to the DNS (basically, it will check the DNS server we configured earlier with Netplan, or the default server provided by DHCP).

There are many other lines in the `nsswitch.conf` file, but I won't discuss them here as they are out of scope of the topic of this section.

The `/etc/hosts` file, which we briefly discussed while working with our hostname, tells our server how to resolve itself (it has a hostname mapping to the localhost IP of `127.0.0.1`), but you are also able to create additional names to IP mappings here as well. For example, if I had a server (`myserver.mydomain.org`) at IP `10.10.96.124`, I could add the following line to `/etc/hosts` to make my machine resolve the server to that IP each time, without it needing to consult a DNS server at all:

```
10.10.96.124 myserver.mydomain.org
```

In practice though, this is usually not a very convenient method by which to configure name resolution. Don't get me wrong, you can certainly list your servers in this file along with their IP addresses, and your server would be able to resolve those names just fine. The problem stems from the fact that this method is difficult to maintain. The name mappings apply only to the server you've made the `/etc/hosts` changes on; other servers wouldn't benefit since they would only check their own `/etc/hosts` file. You could add a list of servers to the hosts file on each of your servers, but that would be a pain to manage. This is the main reason why having a central DNS server is a benefit to any network, especially for resolving the names of local resources.

However, the `/etc/hosts` file is used every now and again in the enterprise as a quick one-off workaround, and you'll probably eventually end up needing to use this method for one reason or another. One very common reason to use such a manual method of resolving names is in the case where you're testing a replacement server. In that case, you can configure the `/etc/hosts` file to have the same name as the original server, but with the IP address of the new server. Once you finish testing and confirm that the new server is operating properly, you can then replace the network-wide DNS name to point to the new IP address.

On legacy Ubuntu servers, the `/etc/resolv.conf` file included the IP addresses for DNS servers the system would use to resolve names. If you wanted to override the DNS servers for your server, you would alter that file. Although this file still exists in Ubuntu 20.04, it's not actually used anymore for this purpose. Name resolution is now handled by `systemd-resolved`, which is a systemd unit that runs in the background and applies DNS settings based on what the system receives via DHCP or what you may have configured in Netplan. For the sake of completeness though, here is a brief overview of the syntax of this file in older releases, in case you end up working on such a server. An example of this file is as follows:

```
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by
resolvconf(8)
# DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
nameserver 10.10.96.1
nameserver 10.10.96.2
```

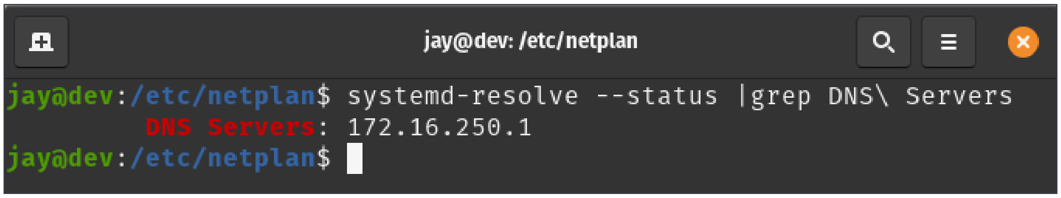
In this example, the `/etc/resolv.conf` output is utilizing servers `10.10.96.1` and `10.10.96.2`. Therefore, the server will first check `/etc/hosts` for a match of the resource you're looking up, and if it doesn't find it, it will then check `/etc/resolv.conf` in order to find out which server to check next. In this case, the server will check `10.10.96.1`.

The `/etc/resolv.conf` file on legacy servers is typically not one that you'd make actual changes to, since it's automatically generated by NetworkManager. NetworkManager is a service that helps you manage your network interfaces, however, it's not used anymore with Ubuntu Server for a handful of releases now. Even though you don't typically manually edit the `/etc/resolv.conf` file, it may be worth looking at on legacy servers to see which DNS servers were assigned, in case you're troubleshooting some sort of networking issue.

Nowadays, modern Ubuntu servers utilize `systemd-resolved` to handle name resolution. If you'd like to see which DNS servers were assigned on a newer Ubuntu Server installation, you can simply look at the configuration file for Netplan that we worked through earlier in the case of a static IP assignment, but if DHCP is being used, the following command will let you know what DNS nameservers your server is currently pointing to:

```
systemd-resolve --status |grep DNS\ Servers
```

This will provide an output similar to that shown in the following screenshot:



```
jay@dev:/etc/netplan$ systemd-resolve --status |grep DNS\ Servers
         DNS Servers: 172.16.250.1
jay@dev:/etc/netplan$
```

Figure 10.5: Viewing a server's current DNS assignment

In a typical enterprise Linux network, you'll set up a local DNS server to resolve your internal resources, which will then forward requests to a public DNS server in case you're attempting to reach something that's not internal. We'll get to that in *Chapter 11*, *Setting Up Network Services*, but you should now understand how the name resolution process works on your Ubuntu Server.

As Linux administrators, we will likely manage a large number of servers, and often the server we're managing may not even be in the same physical location as us. OpenSSH is a powerful tool for remote management, and it's what we'll explore next.

## Getting started with OpenSSH

**OpenSSH** is quite possibly the most useful tool in existence for managing Linux servers. Of all the countless utilities available, this is the one I recommend that everyone starts practicing as soon as they can. Technically, I could probably better fit a section for setting up OpenSSH in *Chapter 11*, *Setting Up Network Services*, but this utility is very handy, and we should start using it as soon as possible.

OpenSSH allows you to open a command shell on other Linux servers, enabling you to run commands as if you were there in front of the server. For Linux administrators like us, this is extremely convenient. We could be tasked with managing dozens, hundreds, or even thousands of servers. With OpenSSH, we can manage our entire server architecture without even getting out of our chair. In this section, I'll give you some information on OpenSSH and how to install it, and then I'll finish up the section with a few examples of actually using it.

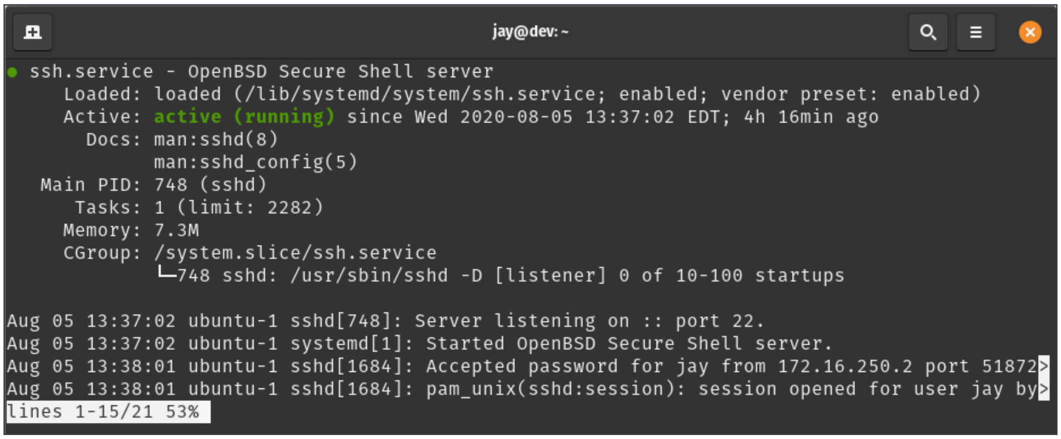
## Installing OpenSSH

OpenSSH consists of two components, the server daemon that runs in the background that accepts SSH connections, and the client that runs on a laptop, workstation, or another server that gives you the ability to connect to an SSH server and run commands. All operating systems nowadays give you access to an OpenSSH client that you can use to make the connection to the server, so that requirement is probably already met. When it comes to Linux, most distributions give you the OpenSSH client already. You can verify that by running `which ssh` at your shell prompt. If you have the client installed, your output should read `/usr/sbin/ssh`.

If, for some reason, you don't have this package installed and you've received no output from the previous command (which would be rare), you can install the OpenSSH client with the following command:

```
sudo apt install openssh-client
```

Depending on the choices you've made during installation, your Ubuntu Server likely has the OpenSSH server installed already. If you don't remember whether or not you opted to have this included during our initial installation, you can run the `which sshd` command at your shell prompt and you should see the output of `/usr/bin/sshd`. You can also execute `systemctl status sshd` as well, and if the server component is present and running, then your server is ready to accept SSH connections:



```
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
     Active: active (running) since Wed 2020-08-05 13:37:02 EDT; 4h 16min ago
       Docs: man:sshd(8)
             man:sshd_config(5)
   Main PID: 748 (sshd)
      Tasks: 1 (limit: 2282)
     Memory: 7.3M
     CGroup: /system.slice/ssh.service
             └─748 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups

Aug 05 13:37:02 ubuntu-1 sshd[748]: Server listening on :: port 22.
Aug 05 13:37:02 ubuntu-1 systemd[1]: Started OpenBSD Secure Shell server.
Aug 05 13:38:01 ubuntu-1 sshd[1684]: Accepted password for jay from 172.16.250.2 port 51872>
Aug 05 13:38:01 ubuntu-1 sshd[1684]: pam_unix(sshd:session): session opened for user jay by>
lines 1-15/21 53%
```

Figure 10.6: Verifying that the OpenSSH service is running on a server

In the case that you don't have the OpenSSH server component installed on your server, you can install it with the following command:

```
sudo apt install openssh-server
```

With great power comes great responsibility, though. As great as OpenSSH is, any service that listens for connections has the potential of being abused. An outside intruder finding a weakness or exploit that allows them to take control over your server is a really bad situation. Therefore, like any service that runs on your server, you should only have it running if you need to use it. Since OpenSSH is incredibly helpful (and it's the standard method of remote management) it's very hard to *not* use it.

There are many methods you can utilize to secure such a service and help protect it. One method is covered in the next section, and we'll talk about security pertaining to OpenSSH again before the book comes to a close. Specifically, in *Chapter 21*, *Securing Your Server*, I will walk you through various configuration changes you can make to help minimize the threat of miscreants breaking into your server from the outside and wreaking havoc.

Securing OpenSSH is actually not hard at all, and would probably only take just a few minutes of your time. Therefore, feel free to make a detour to that chapter to read the section there that talks about securing OpenSSH, and then come back here when you're done. For now, make sure that you have secure, randomly generated passwords on the server at the very least. If OpenSSH is reachable via the public internet, and any of your users have weak passwords, it definitely won't be a fun situation.

With all of that out of the way, we can get started with actually using OpenSSH.

## Issuing commands with OpenSSH

After you've installed the `openssh-server` package on your target machine (the one you want to control remotely), you'll need to start it if it hasn't been already. By default, Ubuntu's `openssh-server` package is automatically configured to start and become enabled once installed. Just as we've done earlier, you can verify that the required service is running with the following command:

```
systemctl status ssh
```

If OpenSSH is running as a daemon on your server, you should see output that tells you that it's `active (running)`. If not, you can start it with the following command:

```
sudo systemctl start ssh
```

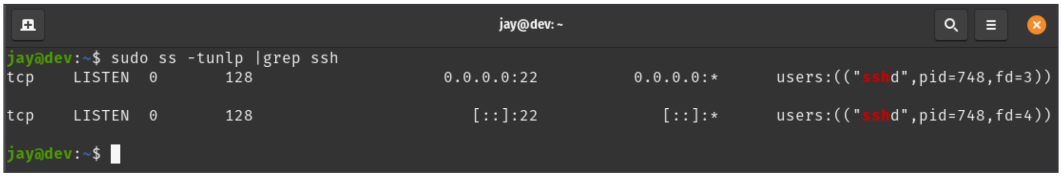
If the output of the `systemctl status ssh` command shows that the daemon is disabled (meaning it doesn't start up automatically when the server boots), you can enable it with the following command:

```
sudo systemctl enable ssh
```

With the OpenSSH server started and running, your server should now be listening for connections. To verify this, use the following command to list listening ports, restricting the output to SSH:

```
sudo ss -tunlp |grep ssh
```

The ss command allows us to view a list of processes running on our server that are listening for connections. It will also display which port a process is listening on. We'll explore this command in more detail in *Chapter 21*, *Securing Your Server*. But for now, this command should produce an output similar to the following:



```
jay@dev:~$ sudo ss -tunlp |grep ssh
tcp    LISTEN  0       128                   0.0.0.0:22          0.0.0.0:*      users:(("sshd",pid=748,fd=3))

tcp    LISTEN  0       128                      [::]:22             [::]:*      users:(("sshd",pid=748,fd=4))

jay@dev:~$
```

Figure 10.7: Checking if the required port for SSH is listening

If, for some reason, your server doesn't show that it has an SSH server listening, double-check that you've started the daemon. By default, the SSH server listens for connections on port 22. This can be changed by modifying the port declaration in the /etc/ssh/sshd_config file, but that's a story for a later chapter. While I won't be going over the editing of this file just yet, just take note that it's the default configuration file for the daemon. OpenSSH reads this file for configuration values each time it's started or restarted.

To connect to a server using SSH, simply execute the ssh command followed by the name or IP address of the server you'd like to connect to:

```
ssh 192.168.1.120
```

By default, the ssh command will use the username you're currently logged in with for the connection. If you'd like to use a different username, specify it with the ssh command by including your username followed by the @ symbol just before the IP address or hostname:

```
ssh fmulder@ 192.168.1.120
```

Unless you tell it otherwise, the ssh command assumes that your target is listening on port 22. If it isn't, you can give the command a different port with the -p option followed by a port number:

```
ssh -p 2242 fmulder@ 192.168.1.120
```

Once you're connected to the target machine, you'll be able to run shell commands and administer the system as if you were right in front of it. You'll have all the same permissions as the user you've logged in with, and you'll also be able to use sudo to run administrative commands if you normally have access to do so on that server.

Basically, anything you're able to do if you were standing right in front of the server, you'll be able to do via SSH. When you're finished with your session, simply type `exit` at the shell prompt, or press *Ctrl* + *d* on your keyboard.

> When you exit an OpenSSH connection, any processes you may have had running in the background will be killed. Be sure you resume any background processes you may have running and finish working with them before you exit your connection. We took a look at how to run processes in the background back in *Chapter 7*, *Controlling and Monitoring Processes*.

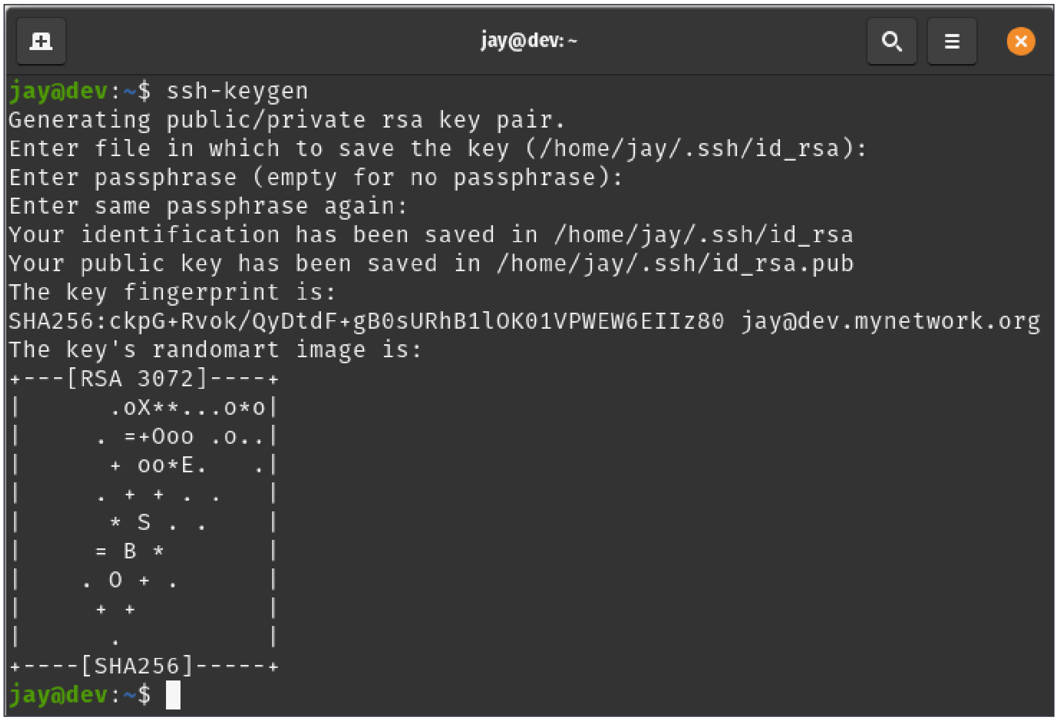
As you can see, OpenSSH is a miraculous tool that will benefit you by allowing you to remotely manage your servers from anywhere you allow SSH access from. Make sure to read the relevant section in *Chapter 11*, *Setting Up Network Services*, with regards to securing it, though. In the next section, we'll discuss SSH key management, which benefits convenience, but also allows you to increase security as well.

## Getting started with SSH key management

When you connect to a host via SSH, you'll be asked for your password, and after you authenticate you'll be connected. Instead of using your password, though, you can authenticate via public key authentication instead. The core benefit to this method is added security, as your system password is never transmitted during the process of connecting to the server. When you create an SSH key-pair, you are generating two files, a *public key* and a *private key*. These two files are mathematically linked, so if you connect to a server that has your public key, it will know it's you because you (and only you) have the private key that matches it. While public key cryptography as a whole is beyond the scope of this book, this method is far more secure than password authentication, and I highly recommend that you use it. To get the most out of the security benefit of authentication via keys, you can actually disable password-based authentication on your server so that your SSH key is your only way in. By disabling password-based authentication and using only keys, you're increasing your server's security by a sizeable margin. We'll go over that in *Chapter 21*, *Securing Your Server*.

## Generating public and private keys

To get started, you'll first need to generate your key. To do so, on your workstation or laptop (the device you're using to connect to the server), use the `ssh-keygen` command as your normal user account. The following screenshot shows what this process generally looks like:



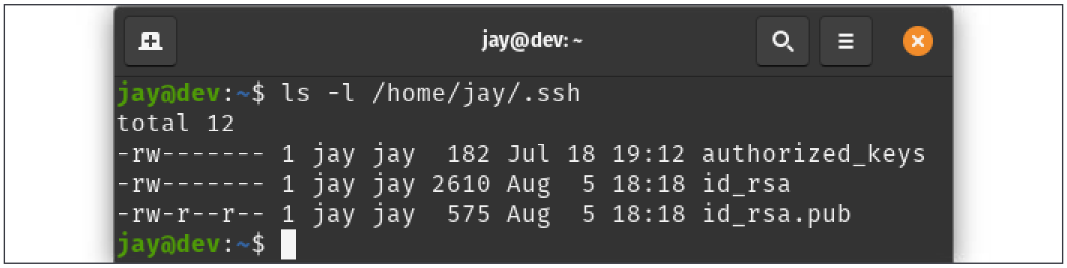
```
jay@dev:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/jay/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/jay/.ssh/id_rsa
Your public key has been saved in /home/jay/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:ckpG+Rvok/QyDtdF+gB0sURhB1lOK01VPWEW6EIIz80 jay@dev.mynetwork.org
The key's randomart image is:
+---[RSA 3072]----+
|      .oX**...o*o|
|     . =+Ooo .o..|
|      + oo*E.   .|
|     . + + . .   |
|      * S . .    |
|     = B *       |
|    . O + .      |
|     + +         |
|      .          |
+----[SHA256]-----+
jay@dev:~$
```

Figure 10.8: Generating an SSH key-pair

First, you'll be asked for the directory in which to save your key files, defaulting to `/home/<user>/.ssh`. You'll next be asked for a passphrase, which is optional. Although it does add an additional step to authenticating via keys, I recommend that you give it a passphrase (which should be different than your system password) since it greatly enhances security. You can press *Enter* for the passphrase without entering one if you do not want this.

What this command does is create a directory named `.ssh` in your `home` directory, if it doesn't already exist. Inside that directory, it will create two files, `id_rsa` and `id_rsa.pub`. The `id_rsa` file is your private key. It should never leave your machine, be given to another user, or be stored on any external media. If your private key leaks out, your keys can no longer be trusted. By default, the private key is owned by the user that created it, with `rw` permissions given only to its owner.

The public key, on the other hand, can leave your computer and doesn't need to be secured as much. Its permissions are more lenient, being readable by everyone and writeable by the owner. You can see this yourself by executing `ls -l /home/<user>/.ssh`:



```
jay@dev:~$ ls -l /home/jay/.ssh
total 12
-rw------- 1 jay jay  182 Jul 18 19:12 authorized_keys
-rw------- 1 jay jay 2610 Aug  5 18:18 id_rsa
-rw-r--r-- 1 jay jay  575 Aug  5 18:18 id_rsa.pub
jay@dev:~$
```

Figure 10.9: Listing the contents of the .ssh directory, showing the permissions of the newly created keys

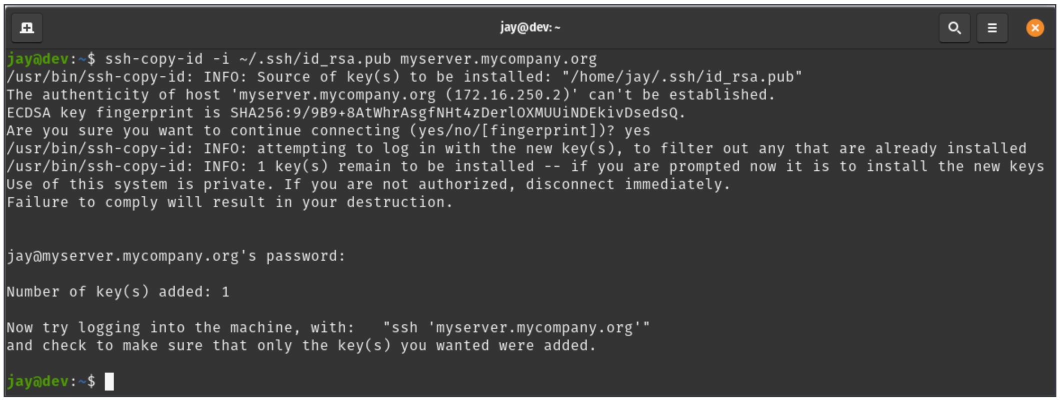
The public key is the key that actually gets copied to other servers to facilitate your being able to log in via such a key-pair. When you log in to a server that has your key, it checks that it's a mathematical match to your private key, and if it is, it will let you log in. You'll also be asked for your passphrase, if you chose to set one when you first created it. But before we can actually use our key, we need to copy it over to the server we intend to connect to.

## Copying your public key to a remote server

To actually transmit your public key to a target server, we can use the `ssh-copy-id` command, as I'm doing in the following example command:

```
ssh-copy-id -i ~/.ssh/id_rsa.pub 192.168.1.150
```

With that command, replace the IP address with the actual IP address, or with the *hostname* of the target server. You'll be asked to log in via password first, and then your key will be copied over. From that point on, you'll log in via your key, falling back to being asked for your password if, for some reason, your key relationship is broken. Here's an example of what this process looks like, if I were to work through the process of copying my key to a server named `myserver.mycompany.org`:



```
jay@dev:~$ ssh-copy-id -i ~/.ssh/id_rsa.pub myserver.mycompany.org
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/jay/.ssh/id_rsa.pub"
The authenticity of host 'myserver.mycompany.org (172.16.250.2)' can't be established.
ECDSA key fingerprint is SHA256:9/9B9+8AtWhrAsgfNHt4zDerlOXMUUiNDEkivDsedsQ.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys
Use of this system is private. If you are not authorized, disconnect immediately.
Failure to comply will result in your destruction.


jay@myserver.mycompany.org's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'myserver.mycompany.org'"
and check to make sure that only the key(s) you wanted were added.

jay@dev:~$
```

Figure 10.10: Using the ssh-copy-id command to copy a public key to a server

So, what exactly did the `ssh-copy-id` command do? Where is your public key copied to, exactly? What happens with this command is that on the target server, a `.ssh` directory is created in your `home` directory on the target server if it didn't already exist. Inside that directory, a file named `authorized_keys` is created if it wasn't already present. The contents of `~/.ssh/id_rsa.pub` on your machine are copied into the `~/.ssh/authorized_keys` file on the target server. With each additional key you add (for example, you connect to that server from multiple machines), the key is added to the end of the `authorized_keys` file, one per line.

> Using the `ssh-copy-id` command is merely a matter of convenience, there's nothing stopping you from copying the contents of your `id_rsa.pub` file and manually pasting it into the `authorized_keys` file of the target server. That method will actually work just fine as well.

When you connect to a server that you have set up a key relationship with by adding your public key, SSH checks the contents of the `~/.ssh/authorized_keys` file on that server, looking for a key that mathematically matches the private key (`~/.ssh/id_rsa`) on your machine. If the two keys are an appropriate match, you are allowed access. If you set up a passphrase, you'll be asked to enter it in order to open your public key.

If you decided not to create a passphrase with your key, you're essentially setting up authentication without a password, meaning you won't be asked to enter anything when authenticating.

## Utilizing an SSH agent

When we created our SSH key earlier, it was mentioned that having a passphrase is optional but is a good idea. Using passphrases with OpenSSH key-pairs boosts their security. If an OpenSSH key falls into the wrong hands, it will be useless if the person attempting to utilize it doesn't know the passphrase. However, we lose a bit of convenience because we need to enter the passphrase for a key each time we want to use it. An OpenSSH key without a passphrase will allow us to connect to a server and be logged in without entering anything at all. With an *SSH agent*, you can actually cache your passphrase the first time you use it, so you won't be asked for it with every connection. This essentially allows you to benefit from the added security of a passphrase, and still maintain at least some convenience. Best of all, if your laptop or desktop is able to utilize the OpenSSH client for connecting to remote systems, you should have an SSH agent on your system already. If we're using a flavor of Linux or macOS on our workstation or laptop, for example, we will have the `ssh-agent` command available to us.

The `ssh-agent` is used by starting it in the background in our terminal. We can then "unlock" our keys with our passphrase, and then the unlocked key will be stored in memory and will be automatically used when we attempt to connect to a server we've copied our public key to. To start it, enter the following command as your normal user account on the machine you're starting your connections from (that is, your workstation):

```
eval $(ssh-agent)
```

This command will start an SSH agent, which will continue to run in the background of your shell. But it's not adding any value to us yet—so we will need to add an SSH key to the now running agent. The `ssh-add` command allows us to add an SSH key to our running `ssh-agent`. To do so, we can give the `ssh-add` command the path to our public key as an argument:

```
ssh-add ~/.ssh/id_rsa
```

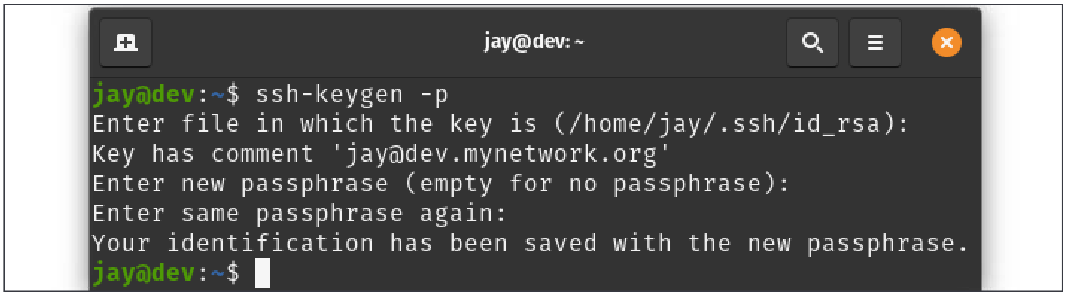
At this point, you'll be asked for your passphrase. As long as you enter it properly, your key will remain open and you won't need to enter it again for future connections, until you close that shell or log out. Now that you have the `ssh-agent` running in the background with your unlocked key, utilizing a key with a passphrase becomes much easier and you'll end up typing a lot less.

## Changing the passphrase of an OpenSSH key

At some point, you may want to change the passphrase associated with a key. If you'd like to do that, you can use the `-p` argument with the `ssh-keygen` command. That same argument can also be used if you didn't choose to add a passphrase when you initially created the key. That's all there is to it. The command you'd enter to add or change a passphrase is as simple as the following:

```
ssh-keygen -p
```

Once you enter that command, press *Enter* to accept the default file (`id_rsa`) unless the key you want to alter is under a different name, in which case you can type the path to, and the name of, that key. Then, you'll be asked for your current passphrase (leave it blank if you don't have one yet) followed by your new passphrase twice. The process looks like this:



```
jay@dev:~$ ssh-keygen -p
Enter file in which the key is (/home/jay/.ssh/id_rsa):
Key has comment 'jay@dev.mynetwork.org'
Enter new passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved with the new passphrase.
jay@dev:~$
```

Figure 10.11: Changing an SSH passphrase

These concepts may take a bit of practice if you've never used SSH before. The best way to practice is to set up multiple Ubuntu Server installations (perhaps several virtual machines), and practice using SSH to connect to them, as well as deploying your key to each machine via the `ssh-copy-id` command. It's actually quite easy once you get the hang of it.

## Simplifying SSH connections with a config file

Before we leave the topic of OpenSSH, there's another trick that benefits convenience, and that is the creation of a local configuration file for SSH. This file must be stored in the `.ssh` directory of your home directory, and be named `config`. The full path for this file in my case looks like this:

```
/home/jay/.ssh/config
```

This file doesn't exist by default, but if it's found, SSH will parse it whenever you use the client and you'll be able to benefit from it. Go ahead and open this file in your text editor, such as `nano`:

```
nano /home/your_username/.ssh/config
```

This `config` file allows you to type configuration for servers that you connect to often, which can simplify the `ssh` command automatically. The following are example contents from such a file that will help me illustrate what it does:

```
host myserver
    Hostname 192.168.1.23
    Port 22
    User jdoe

Host nagios
    Hostname nagios.mynetwork.org
    Port 2222
    User nagiosuser
```

In the example contents, I have two hosts outlined, `myserver` and `nagios`. For each, I've identified a way to reach it by name or IP address (the `Hostname` line), as well as the `Port` and `User` account to use for the connection. If I use `ssh` to connect to either `Host` by the name I outlined in this file, it will use the values I have stored there, for example:

```
ssh nagios
```

That command is a lot shorter than if I set all the options manually. Considering I have a `config` file for SSH, that command is essentially the same as if I identified the connection details manually, which would've been the following:

```
ssh -p 2222 nagiosuser@nagios.mynetwork.org
```

I'm sure you can see how much simpler it is to type the first command than the second. With the `config` file for SSH, I can have some of those details automatically applied. Since I've outlined that my `nagios` server is located at `nagios.mynetwork.org`, its SSH user is `nagiosuser`, and it's listening on port `2222`, it will automatically use those values even though I only typed `ssh nagios`. Furthermore, you can also override this entry as well. If you provide a different username when you use the `ssh` command, it will use that instead of what you have written in the `config` file.

In the first example (for the `myserver` server), I'm providing an IP address for the connection, rather than a hostname. This is useful in a situation where you may not have a DNS entry for your target server. With this example, I don't have to remember that the IP address for `myserver` is `192.168.1.23`. I simply execute `ssh myserver` and it's taken care of for me.

The names of each server within the `config` file are arbitrary, and don't have to match the target server's hostname. I could've named the first server `potato` and it would still have routed me to `192.168.1.23`, so I can create any sort of named shortcut I want, whatever I find is most convenient for me and easiest to remember. As you can see, maintaining a `config` file in your home directory containing your most commonly used SSH connections will certainly help keep you organized and allow you to connect more easily.

## Summary

In this chapter, we worked through several examples of connecting to other networks. We started off by configuring our hostname, managing network interfaces, assigning static IP addresses, as well as looking at how name resolution works in Linux. A decent portion of this chapter was dedicated to topics regarding OpenSSH, which is an extremely useful utility that allows you to remotely manage your servers. We'll revisit OpenSSH in *Chapter 21*, *Securing Your Server*, with a look at boosting its security. Overall, we've only begun to scratch the surface of this tool. Entire books have been written about SSH, but the examples in this chapter should be enough to make you productive with it. The name of the game is to practice, practice, practice!

In the next chapter, we'll talk about managing software packages. We'll work through adding and removing them, adding additional repositories, and more!

## Further reading

- **Netplan FAQ**: `https://netplan.io/faq`
- **Ubuntu SSH key documentation**: `https://help.ubuntu.com/community/SSH/OpenSSH/Keys`
- **NetworkInterfaceNames** (from Debian's wiki): `https://wiki.debian.org/NetworkInterfaceNames`

# 11
# Setting Up Network Services

In *Chapter 10, Connecting to Networks*, we went over some important foundational topics related to networking. We saw how to set the hostname, manage network interfaces, configure connections, and more. In this chapter, we'll revisit networking, specifically to set up the resources that will serve as the foundation of our network. The majority of this chapter will focus on setting up the DHCP and DNS servers, which are very important components of any network. In addition, we'll also set up a **Network Time Protocol** (**NTP**) server to keep our clocks synchronized. We'll even take a look at setting up a server to act as an internet gateway for the rest of our network.

Along the way, we'll cover the following topics:

- Planning your IP address scheme
- Serving IP addresses with `isc-dhcp-server`
- Setting up DNS with `bind`
- Setting up an internet gateway
- Keeping your clock in sync with NTP

As a Linux administrator, you may or may not be tasked with designing the entire network layout of your organization; often there will be an already-existing network to manage. In the next section, we'll discuss creating such a layout, even if only to understand what goes into such a plan.

## Planning your IP address scheme

Designing the overall layout of your network is an incredibly important process that sets the stage for success or failure later on. This design must take into consideration the needs of the organization, the need for efficient methods of communication, and the segregation of network services to ensure that your servers can only communicate with the resources that they are supposed to. As a Linux administrator that manages a fleet of Ubuntu servers, it's not always the case that you'll even have a say in the network layout at all. It's quite common that you "inherit" a network designed by a previous administrator, or your job is siloed such that you only manage the servers and someone else is responsible for the network.

Since this is primarily a book that focuses on teaching you how to manage Ubuntu servers, we aren't going to cover all of the things that a network administrator would need to know, but there's quite a bit of common knowledge between the two roles. As a Linux administrator, you may or may not be tasked with the designing of your network, but at a minimum, you do need to understand the overall layout and how your servers will fit within it.

In this section, we'll discuss the most important part of a network layout—the IP address scheme. Planning the IP scheme is an important task that sets the foundation for many other things. Even if it's not up to you to design this layout, knowing the finer details can help you understand the weaknesses when something goes wrong. Planning your IP layout involves estimating how many devices will need to connect to your network and being able to support them. In addition, a good plan will account for potential growth and allow expansion as well. The main thing that factors into this is the size of your user base. Perhaps you are working in a small office with only a handful of people, or a large corporation with thousands of users and hundreds of virtual machines. Even if your organization is only a small office, I always recommend making the assumption that your company will explode in popularity someday and designing your network to have the growth potential to accommodate that.

Typically, most off-the-shelf routers and network equipment come with an integrated **Dynamic Host Control Protocol** (**DHCP**) server, with a default class C (`/24`) network. Essentially, this means that if you do not perform any configuration at all, you're limited to 254 addresses. For a small office, this may seem like plenty. After all, if you don't even have 254 employees at your company, that number may seem like overkill. As I mentioned before, potential growth is always something to keep in mind. But even if we remove that from the equation, IP addresses are used up quicker than you'd think nowadays—even when it comes to internal addressing. An average user may consume three IP addresses each, and sometimes more.

For example, perhaps a user not only has a laptop (which itself can have both a wired and wireless interface, both consuming an IP address), but perhaps they also have a mobile phone (which likely features Wi-Fi), and a **Voice over IP** (**VoIP**) phone (there goes another address). If that user somehow manages to convince their supervisor that they also need a desktop computer as well as their laptop, there will be a total of five IP addresses for that one user. Suddenly, 254 addresses doesn't seem like all that many.

Perhaps a really good real-world example of this is a small restaurant in a busy city that offers free Wi-Fi to its customers. The person designing the guest network for their customers may assume that a `/24` network is more than enough, if they only have a hundred customers or so each day. While that logic may seem sound, consider that if the restaurant is next to a busy street, people that have Wi-Fi enabled on their phone may snag an address as they pass by, and if the DHCP addresses are configured to expire in 24 hours, then there will often be no available IP addresses at all most of the time. If you've ever attempted to connect to a restaurant Wi-Fi network and found yourself unable to access the internet after connecting, this could be a real-world example of a pain-point of not designing a network layout effectively.

The obvious answer to this problem is splitting up your network into **subnets**. Although I won't go into the details of how to subnet your network (which would be beyond the scope of this book), I mention it here because it's definitely something you should take into consideration. In the next section, I'll explain how to set up your own DHCP server with a single network. However, if you need to expand your address space, you can easily do so by updating your DHCP configuration. When coming up with an IP address layout, always assume the worst and plan ahead. While it may be a simple task to expand your DHCP server, planning a new IP scheme rollout is very time consuming, and to be honest, annoying.

When I set up a new network, I like to divide the address space into several categories. First, I'll usually set aside a group of IP addresses specifically for DHCP. These addresses will get assigned to clients as they connect, and I'll usually have them expire and need to be renewed in about one day. Then, I'll set aside a block of IP addresses for network appliances, another block for servers, and so on. In the case of a typical `/24`, I might decide on a scheme such as the following (assuming it's a small office with no growth planned):

```
Network: 192.168.1.0/24
Network equipment: 192.168.1.1 - 192.168.1.10
Servers: 192.168.1.11 - 192.168.1.99
DHCP: 192.168.1.100 - 192.168.1.240
Reservations: 192.168.1.241 - 192.168.1.254
```

Since I mentioned it's a good idea to plan for future growth, the `/24` would be potentially constrained and wouldn't accommodate much growth. I chose that scheme to keep everything in this chapter simple for the sake of easy explanation. But in a real company network, you may want to consider a larger number of IP addresses than what the above would give you.

How do you get more IP addresses? Take a look at the number after the network address, which is `/24` on the first line. With that number, we're configuring how large the IP address space is, which essentially correlates to how many IP addresses we have available. That number is known as a **CIDR** notation, which stands for **Classless Inter-Domain Routing**. Similar to a subnet mask, changing that number results in a different number of IP addresses. For example, if you change the `/24` portion of the network to `/22`, you instantly have 1,022 possible addresses to work with, rather than 254. If you lower it again to `/20`, the number jumps to 4,094. For now, don't worry too much about this. You can focus on remembering that the higher the number, the fewer IP addresses you have. It can go up to `/32`, which only gives you one IP address. If you'd like to experiment further with subnetting, we've provided a subnet calculator in the *Further reading* section.

Of course, no single IP address scheme is right for everyone. The one I provided is simply a hypothetical example, so you shouldn't copy mine and use it on your network unless it matches your needs. I'll use this scheme for the remainder of this chapter, since it works fine as an example. To explain my sample rollout, we start off with a 24-bit network, `192.168.1.0/24`. The address `192.168.1.0` refers to the network itself, and that IP address is not assignable to clients. The first usable IP address in this subnet will be `192.168.1.1`. The last IP address in this block (`192.168.1.255`) is not assignable either, since that is known as the **broadcast address**. Anything that's sent to the broadcast address is effectively sent to every IP in the block, so we can't really use it for anything but broadcasts. In summary, keep in mind that an IP address ending in `.0` can't be used, and neither can an IP address that ends in `.255`.

The following is a list of common CIDR values and their impact on the number of available IP addresses:

| **CIDR** | **Total Usable IP Addresses** |
|---|---|
| `/32` | 1 |
| `/24` | 254 |
| `/16` | 65,534 |
| `/8` | 16,777,214 |

Going back to the example IP layout mentioned earlier in this section, I set aside a group of IP addresses starting with `192.168.1.1` through `192.168.1.10` for use by network appliances. Typical devices that would fit into this category would be managed switches, routers, wireless access points, and so on. These devices typically have an integrated web console for remote management, so it would be best to have a static IP address assignment. That way, I'll have an IP address available that I can use to access these devices. I like to set up network appliances as the first devices so that they all get the lowest numbers when it comes to the last number of each IP address. This is just personal preference.

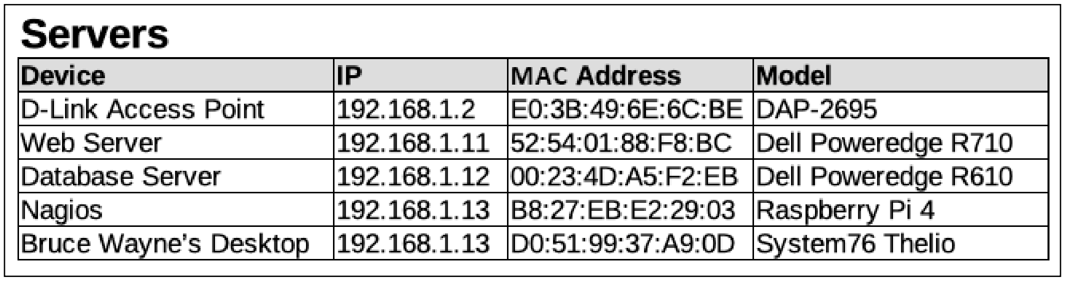
Next in the example layout, we define IP addresses `192.168.1.11` through `192.168.1.99` for servers. This may seem like quite a few addresses for servers, and it is. However, with the rise of virtualization and how simple it has become to spin up a server, this block could get used up faster than you'd think. Feel free to adjust accordingly.

Now we have our DHCP pool, which consists of addresses `192.168.1.101` through `192.168.1.240`. These IP addresses are assignable to any devices that connect to our network. Typically, I like to have these assignments expire in one day to prevent one-off devices from claiming and holding onto an IP address for too long, which can lead to devices fighting over a DHCP lease. In this situation, you'd have to clear your DHCP leases to reset everything, and I find that to be too much of a hassle. When we get to the section on setting up a DHCP server, I'll show you how to set the expiration time.

Finally, we have addresses `192.168.1.241` through `192.168.1.254` for the purposes of DHCP reservations. I generally refer to reserved DHCP addresses as *static leases*, but both terms mean the same thing. These addresses will be assigned by DHCP, but each device with a static lease will be given the same IP address each time. You don't have to separate these into their own pool, since DHCP will not assign the same address twice. It may still be a good idea to separate them though, if only to be able to tell from looking at an IP address that it's a static lease, due to it being within a particular hypothetical block. Static leases are good for devices that aren't necessarily a server, but still need a predictable IP address. An example of this may be an administrator's desktop PC. Perhaps they want to be able to connect to the office via VPN and be able to easily find their computer on the network and connect to it. If the IP was dynamically assigned instead of statically assigned, it would be harder for them to find it.

After you carve up your IP addresses, the next thing is to make sure that they're accurately documented. If you don't focus on documentation now while designing services, you will definitely regret it later. You can consider setting up a private Wiki server, for example.

Another method is creating a spreadsheet to keep track of your static IP assignments. This is acceptable if you don't have a better solution; it doesn't have to be anything fancy. Among the usual components, such as the device info and IP address, I also include the MAC address of each device on the spreadsheet, which will come in handy when we set up our DHCP server in the next section:



**Servers**

| Device | IP | MAC Address | Model |
|---|---|---|---|
| D-Link Access Point | 192.168.1.2 | E0:3B:49:6E:6C:BE | DAP-2695 |
| Web Server | 192.168.1.11 | 52:54:01:88:F8:BC | Dell Poweredge R710 |
| Database Server | 192.168.1.12 | 00:23:4D:A5:F2:EB | Dell Poweredge R610 |
| Nagios | 192.168.1.13 | B8:27:EB:E2:29:03 | Raspberry Pi 4 |
| Bruce Wayne's Desktop | 192.168.1.13 | D0:51:99:37:A9:0D | System76 Thelio |

Figure 11.1: An example IP address layout spreadsheet

Although subnetting is beyond the scope of this book, it's definitely something you should look into if you're not already familiar with it. As you can see from my example layout, our number of available addresses is rather limited with a 24-bit network. However, this layout will serve as an example we can follow that's good enough for the remainder of the chapter. At this point, just think about the factors that are important to your organization, and ensure that any networks you create are scalable and accommodate your needs.

Now that we have an IP layout (if we didn't have one already), we can take a look at setting up a DHCP server, which will be the service that will ultimately be assigning these IPs.

## Setting up a DHCP server for serving IP addresses

Most network appliances you purchase nowadays often come with their own DHCP server, and allow you to configure it via a web console. Often, this is totally fine and meets your needs. In my experience though, network appliances you purchase can be very hit or miss. Some of them are great, while others are not so impressive. One of the main problems is that manufacturers will often stop supporting the hardware prematurely, exposing your network to unpatched vulnerabilities. It's important to always purchase network hardware that's future-proof. Or, you can set up your own router with the features that you need. Although the burden of management is placed on you, this gives you ultimate flexibility. Ubuntu servers make great DHCP servers, and rolling your own server is actually easier than it sounds. And that's exactly what we're going to work through in this section.

First, the server that serves DHCP will definitely need a static IP address. This means you'll need to configure Netplan with a static IP assignment. A static lease won't work here, since the DHCP server can't assign an IP address to itself. Also, the IP address that you designate for your server's static IP must be in the same network as the addresses that you plan to serve. Otherwise, the service will fail to start even after we configure it.

> If you have yet to set a static IP address, *Chapter 10*, *Connecting to Networks*, has a section that will walk you through the process.

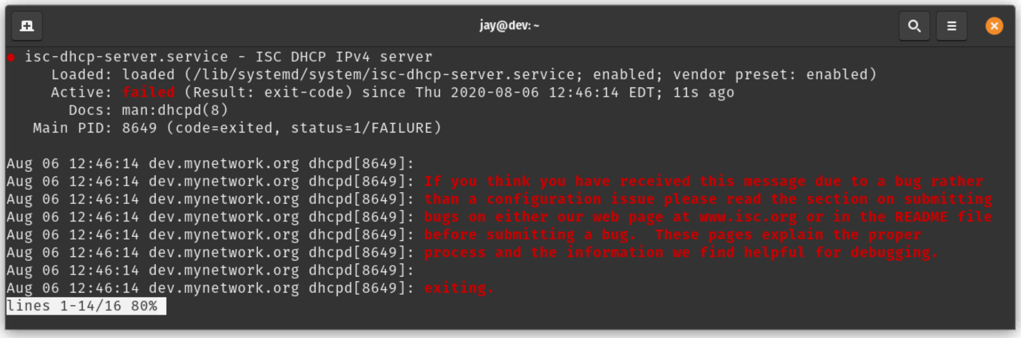
Once you assign a static IP address, the next step is to install the `isc-dhcp-server` package:

```
sudo apt install isc-dhcp-server
```

Check the status of the daemon after installing the `isc-dhcp-server` package, using the following command:

```
systemctl status isc-dhcp-server
```

You'll likely notice that it failed:



```
● isc-dhcp-server.service - ISC DHCP IPv4 server
     Loaded: loaded (/lib/systemd/system/isc-dhcp-server.service; enabled; vendor preset: enabled)
     Active: failed (Result: exit-code) since Thu 2020-08-06 12:46:14 EDT; 11s ago
       Docs: man:dhcpd(8)
   Main PID: 8649 (code=exited, status=1/FAILURE)

Aug 06 12:46:14 dev.mynetwork.org dhcpd[8649]:
Aug 06 12:46:14 dev.mynetwork.org dhcpd[8649]: If you think you have received this message due to a bug rather
Aug 06 12:46:14 dev.mynetwork.org dhcpd[8649]: than a configuration issue please read the section on submitting
Aug 06 12:46:14 dev.mynetwork.org dhcpd[8649]: bugs on either our web page at www.isc.org or in the README file
Aug 06 12:46:14 dev.mynetwork.org dhcpd[8649]: before submitting a bug.  These pages explain the proper
Aug 06 12:46:14 dev.mynetwork.org dhcpd[8649]: process and the information we find helpful for debugging.
Aug 06 12:46:14 dev.mynetwork.org dhcpd[8649]:
Aug 06 12:46:14 dev.mynetwork.org dhcpd[8649]: exiting.
lines 1-14/16 80%
```

Figure 11.2: isc-dhcp-server failing by default

If it did fail to start, there's no need to be concerned. Ubuntu, by default, starts most of the services that are installed via packages. Sometimes, a service needs to be configured before it can run. In the case of the `isc-dhcp-server`, it needs a valid configuration in order to start, but we have yet to configure anything. We need to configure the `isc-dhcp-server` service for it to be useful, so let's stop the service for now.

We can start it as soon as we've finished adding our configuration:

```
sudo systemctl stop isc-dhcp-server
```

Configuring an IPv6 network is beyond the scope of this chapter, but the DHCP server package we've just installed also comes with an IPv6 equivalent. Let's stop and disable this service, since we won't be using it:

```
sudo systemctl stop isc-dhcp-server6
sudo systemctl disable isc-dhcp-server6
```

Now that you've installed the `isc-dhcp-server` package, you'll have a default configuration file for it at `/etc/dhcp/dhcpd.conf`. This file will contain some default configuration, with some example settings that are commented out. Feel free to take a look at this file to get an idea of some of the settings you can configure. We'll create our own `dhcpd.conf` file from scratch. So when you're done looking at it, copy the existing file with a new name so we can refer to it later if we ever need to:

```
sudo mv /etc/dhcp/dhcpd.conf /etc/dhcp/dhcpd.conf.orig
```

Now, we're ready to create our own `dhcpd.conf` file. Open `/etc/dhcp/dhcpd.conf` in your preferred text editor. Since the file no longer exists (we moved it), we should start with an empty file. Here's an example `dhcpd.conf` file that I will explain so that you understand how it works:

```
default-lease-time 43200;
max-lease-time 86400;
option subnet-mask 255.255.255.0;
option broadcast-address 192.168.1.255;
option domain-name "local.lan";
authoritative;
subnet 192.168.1.0 netmask 255.255.255.0 {
    range 192.168.1.100 192.168.1.240;
    option routers 192.168.1.1;
    option domain-name-servers 192.168.1.1;
}
```

As always, change the values I've used to those that match your network. I'll explain each line so that you'll understand how it affects the configuration of your DHCP server.

```
default-lease-time 43200;
```

When a device connects to your network and requests an IP address, the expiration of the lease will be set to the number of seconds in `default-lease-time` if the device doesn't explicitly ask for a longer lease time. Here, I'm setting that to `43200` seconds, which is equivalent to half a day. This basically means that the device will need to renew its IP address every `43200` seconds, unless it asks for a longer duration.

```
max-lease-time 86400;
```

While the previous setting dictated the default lease time for devices that don't ask for a specific lease time, `max-lease-time` is the most that the device is allowed to have. In this case, I set this to one day (`86400` seconds). Therefore, no device that receives an IP address from this DHCP server is allowed to hold onto their lease for longer than this without first renewing it.

```
option subnet-mask 255.255.255.0;
```

With this setting, we're informing clients that their subnet mask should be set to `255.255.255.0`, which is for a default 24-bit network. If you plan to subnet your network, you'll put in a different value here. `255.255.255.0` is fine if all you need is a 24-bit network.

```
option broadcast-address 192.168.1.255;
```

With this setting, we're telling the client to use `192.168.1.255` as the broadcast address, which is the last address in the subnet and cannot be assigned to a host.

```
option domain-name "local.lan";
```

Here, we're setting the domain names of all hosts that connect to the server to include `local.lan`. The domain name is added to the end of the hostname. For example, if a workstation with a hostname of `muffin` receives an IP address from our DHCP server, it will be referred to as `muffin.local.lan`. Feel free to change this to the domain name of your organization, or you can leave it as is if you don't have one.

```
authoritative;
```

With the `authoritative;` setting (the opposite is `not authoritative;`), we're declaring our DHCP server as authoritative to our network. Unless you are planning to have multiple DHCP servers, the `authoritative;` option should be included in your `config` file. We won't use the `non authoritative;` option as it's beyond the scope of this chapter.

Now, we get to the most important part of our configuration file for DHCP. The following block details the specific information that will be provided to clients:

```
subnet 192.168.1.0 netmask 255.255.255.0 {
    range 192.168.1.100 192.168.1.240;
    option routers 192.168.1.1;
    option domain-name-servers 192.168.1.1;
}
```

This block is probably self-explanatory, but we're basically declaring our pool of addresses for the `192.168.1.0` network. We're declaring a range of IPs from `192.168.1.100` through `192.168.1.240` to be available from clients. Now when our DHCP server provides an address to clients, it will choose one from this pool. For the address pool (`range`), feel free to expand it or shrink it accordingly. For example, you might need more addresses than the 140 that are allowed in my sample range, so you may change it to something like `192.168.1.50` through `192.168.1.250`. Feel free to experiment.

We're also providing a default gateway (`option routers`) and DNS server (`option domain-name-servers`) of `192.168.1.1`. This is assuming that your router and local DNS server are both listed at that address, so make sure that you change it accordingly. Otherwise, anyone who receives a DHCP lease from your server will not be able to connect to anything.

Now we have our configuration file in place, but the DHCP server will likely still not start until we declare an interface for it to listen for requests on. You can do that by editing the `/etc/default/isc-dhcp-server` file, where you'll see a line toward the bottom similar to the following:

```
INTERFACESv4=""
```

Simply type the name of the interface within the quotes:

```
INTERFACESv4="enp0s3"
```

In case you forgot, the command to list the details of the interfaces on your server is `ip addr show`, or the shortened version, `ip a`.

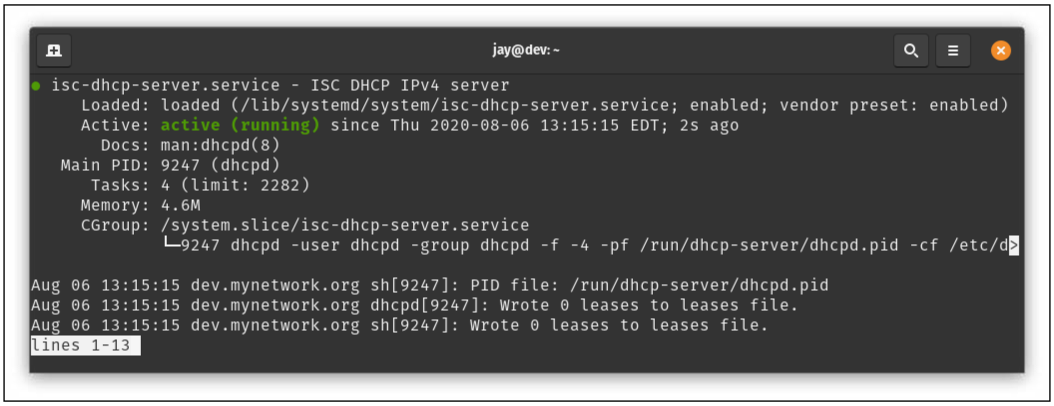
Now that we have our DHCP server configured, we should be able to start it:

```
sudo systemctl start isc-dhcp-server
```

Next, double-check that there were no errors by running the following command:

```
sudo systemctl status isc-dhcp-server
```

The daemon should report that it's `active (running)`, similar to what's shown in the following screenshot:



```
● isc-dhcp-server.service - ISC DHCP IPv4 server
     Loaded: loaded (/lib/systemd/system/isc-dhcp-server.service; enabled; vendor preset: enabled)
     Active: active (running) since Thu 2020-08-06 13:15:15 EDT; 2s ago
       Docs: man:dhcpd(8)
   Main PID: 9247 (dhcpd)
      Tasks: 4 (limit: 2282)
     Memory: 4.6M
     CGroup: /system.slice/isc-dhcp-server.service
             └─9247 dhcpd -user dhcpd -group dhcpd -f -4 -pf /run/dhcp-server/dhcpd.pid -cf /etc/d>

Aug 06 13:15:15 dev.mynetwork.org sh[9247]: PID file: /run/dhcp-server/dhcpd.pid
Aug 06 13:15:15 dev.mynetwork.org dhcpd[9247]: Wrote 0 leases to leases file.
Aug 06 13:15:15 dev.mynetwork.org sh[9247]: Wrote 0 leases to leases file.
lines 1-13
```

Figure 11.3: Checking the status of the isc-dhcp-server process after a successful start

Assuming all went well, your DHCP server should be running. When an IP lease is assigned to a client, it will be recorded in the `/var/lib/dhcp/dhcpd.leases` file. While your DHCP server runs, it will also record information to your server's system log, located at `/var/log/syslog`. To see your DHCP server function in all its glory, you can follow the log as it gets written to with the following:

```
sudo tail -f /var/log/syslog
```

We've discussed the `-f` flag of the `tail` command earlier in the book. This option is indispensable, and it's something you'll likely use quite often as a server administrator. With the `-f` option, you'll watch the log as it gets written to, rather than needing to refresh it manually. You can press *Ctrl* + *c* to break out of the file.

While your DHCP server runs, you'll see notices appear within the `syslog` file whenever a DHCP request was received and when a lease is offered to a client. A typical DHCP request will appear in the log similar to the following:

```
Oct  5 22:07:36 myserver dhcpd: DHCPDISCOVER from 52:54:00:88:f8:bc via
enp0s3
Oct  5 22:07:36 myserver dhcpd: DHCPOFFER on 192.168.1.103 to
51:52:01:87:f7:bc via enp0s3
```

Active and previous DHCP leases are stored in the `/var/lib/dhcp/dhcpd.leases` file, and a typical lease entry in that file would look similar to the following:

```
lease 192.168.1.138 {
  starts 0 2020/10/05 16:37:30;
  ends 0 2020/10/06 16:42:30;
  cltt 0 2020/20/06 16:37:30;
  binding state active;
  next binding state free;
  rewind binding state free;
  hardware ethernet 32:6e:92:01:1f:7f;
}
```

When a new device is added to your network and receives an IP address from your new DHCP server, you should see the lease information populate that file. This file can be incredibly helpful, because whenever you connect a new device, you won't have to interrogate the device itself to find out what its IP address is. You can just check the `/var/lib/dhcp/dhcpd.leases` file. If the device advertises its hostname, you'll see it within its lease entry. A good example of how this can be useful is connecting a Raspberry Pi to your network. Once you plug it in and turn it on, you'll see its IP address in the `dhcpcd.leases` file, and then you can connect to it via SSH with the IP without having to plug a monitor into it to find out which IP it was given. Similarly, you can view the temporary IP address of a new network appliance that you plug in so that you can connect to it and configure it.

If you have any trouble setting up the `isc-dhcp-server` daemon, double-check that you have set all the correct and matching values within your static IP assignment, as well as within your `/etc/dhcp/dhcpd.conf` file. For example, your server must be within the same network as the IPs you're assigning to clients. As long as everything matches, you should be fine and it should start properly.

Next, let's take a look at another important service within a network—DNS.

## Adding a DNS server

I'm sure most of you are familiar with the purpose of a **Domain Name System** (**DNS**) server. Its simplest definition is that it's a service that's responsible for matching an IP address to a domain or hostname. When you connect to the internet, name-to-IP matching happens constantly as you browse. After all, it's much easier to connect to `https://www.google.com/` using its domain name than it is to remember its IP address. When you connect to the internet, your workstation or server will connect to an external DNS server in order to figure out the IP addresses for the websites you attempt to visit.

It's also very common to run a local DNS server internally in your organization. The benefit is that you'll be able to resolve your local hostnames as well, something that an external DNS server would know nothing about. For example, if you have an intranet site that you intend to make available to your co-workers, it would be easier to give everyone a local domain that they can access than it would be to make everyone memorize its IP address. With a local DNS server, you would create what is known as a **zone file**, which would contain information regarding the hosts and IP addresses in use within your network so that local devices would be able to resolve them. In the event that your local DNS server is unable to fulfill your request (such as a request for an external website), the server would pass the request along to an external DNS server, which would then carry out the request.

A detailed discussion of DNS, how it functions, and how to manage it is outside the scope of this book. However, a basic understanding is really all you need in order to make use of a DNS server within your network. In this section, I'll show you how to set up your very own DNS server to allow your devices to resolve local hostnames, which should greatly enhance your network.

## Setting up external DNS with bind

To set up your very own DNS server, we'll first need to install the **Berkeley Internet Name Daemon** (**BIND**) package on our server:

```
sudo apt install bind9
```

Now, we should have the `bind9` service running on our server, though it's not actually configured to do much at this point. The most basic function of `bind` is to act as what's called a **Caching Name Server**, which means that the server doesn't actually match any names itself. Instead, it caches responses from an external server. We'll configure `bind` with actual hosts later, but setting up a caching name server is a good way to get started.

To do so, open the `/etc/bind/named.conf.options` file in your favorite text editor.

Within the file, you should see a block of text that looks similar to the following:

```
// forwarders {
//      0.0.0.0;
// };
```

Uncomment these lines. The *forward slashes* are comment marks as far as this configuration file is concerned, so remove them. Then, we can add a few external DNS server IP addresses. For these, you can use the IP addresses for your **Internet Service Provider** (**ISP**)'s DNS servers, or you could simply use Google's DNS servers (`8.8.8.8` and `8.8.4.4`) instead:

```
forwarders {
  8.8.8.8;
  8.8.4.4;
};
```

After you save the file, restart the `bind9` service:

```
sudo systemctl restart bind9
```

To be sure that everything is running smoothly, check the status of the service:

```
systemctl status bind9
```

It should report that it's `active (running)`. As long as you've entered everything correctly, you should now have a working DNS server. Of course, we haven't added any DNS names for it to resolve, but we'll get to that. Now, all you should need to do is configure other devices on your network to use your new DNS server. The easiest way to do this is to reconfigure the `isc-dhcp-server` service we set up in the previous section. Remember the section that designates a pool of addresses from the server to the clients? It also contained a section to declare the DNS server your clients will use as well. Here's that section again, with the relevant line in bold:

```
subnet 192.168.1.0 netmask 255.255.255.0 {
  range 192.168.1.100 192.168.1.240;
  option routers 192.168.1.1;
  option domain-name-servers 192.168.1.1;
}
```

To configure the devices on your network to use your new DNS server, all you should need to do is change the configuration `option domain-name-servers 192.168.1.1;` to point to the IP address of the new DNS server that we're in the process of setting up. When clients request a DHCP lease (or attempt to renew an existing lease), they will be configured with the new DNS server automatically.

With the caching name server we just set up, hosts that utilize it will check it first for any hosts they attempt to look up. If they look up a website or host that is not within your local network, their requests will be forwarded to the forwarding addresses you configured for `bind`.

In my example, I used Google's DNS servers, so if you used my configuration, your hosts will first check your local server and then check Google's servers when resolving external names. Depending on your network hardware and configuration, you might even see a slight performance boost. This is because the DNS server you just set up is caching any lookups done against it. For example, if a client looks up `https://www.packtpub.com` in a web browser, your DNS server will forward the request along since that site doesn't exist locally and it will also remember the result. The next time a client within your network looks up that site, the response will be much quicker because your DNS server cached it.

To see this yourself, execute the following command twice on a node (device) that is utilizing your new DNS server:

```
dig www.packtpub.com
```

If `dig` isn't available in your installation, you can install it as part of the `dnsutils` package:

```
sudo apt install dnsutils
```

In the response, look for a line toward the end that gives you your query time. It will look similar to the following:

```
;; Query time: 98 msec
```

When you run it again, the query time should be much lower:

```
;; Query time: 1 msec
```

This is your caching name server in action! Even though we haven't even set up any zone files to resolve your internal servers, your DNS server is already adding value to your network. You just laid the groundwork we'll use for the rest of our configuration.

## Setting up internal DNS and adding hosts

Now, let's add some hosts to our DNS server so we can start fully utilizing it. The configuration file for `bind` is located at `/etc/bind/named.conf`. In addition to some commented lines, it will have the following three lines of configuration within it:

```
include "/etc/bind/named.conf.options";
include "/etc/bind/named.conf.local";
include "/etc/bind/named.conf.default-zones";
```

As you can see, the default `bind` configuration is split among several configuration files. Here, it includes three others: `named.conf.options`, `named.conf.local`, and `named.conf.default-zones` (the first of which we already took care of editing). In order to resolve local names, we need to create a **zone file**, which is essentially a text file that includes some configuration, a list of hosts, and their IP addresses. In order to do this, we need to tell `bind` where to find the zone file we're about to create. Within `/etc/bind/named.conf.local`, we need to add a block of code like the following to the end of the file:

```
zone "local.lan" IN {
    type master;
    file "/etc/bind/net.local.lan";
};
```

Notice that the zone is named `local.lan`, which is the same name I gave our domain in our DHCP server configuration. It's best to keep everything consistent when we can. If you use a different domain name than the one I used in my example, make sure that it matches here as well. Within the block, we're creating a `master` zone file and informing `bind` that it can find a file named `net.local.lan`, stored in the `/etc/bind` directory. This should be the only change we'll need to make to the `named.conf.local` file; we'll only create a single zone file (for the purpose of this section). Once you save this file, you'll need to create the `/etc/bind/net.local.lan` file. So, go ahead and open that file in a text editor. Since we haven't created it yet, it should be blank. Here's an example of this zone file, completely filled out with some sample configuration:

```
$TTL 1D
@ IN SOA local.lan. hostmaster.local.lan. (

202008161; serial

8H ; refresh
4H ; retry
4W ; expire
1D ) ; minimum
IN A 192.168.1.1
;
@ IN NS hermes.local.lan.
fileserv        IN  A   192.168.1.3
hermes          IN  A    192.168.1.1
mailserv        IN  A   192.168.1.5
mail            IN  CNAME   mailserv.
web01           IN  A   192.168.1.7
```

Feel free to edit this file to match your configuration. You can edit the list of hosts at the end of the file to match your hosts within your network, as the ones I included are merely examples. You should also ensure that the file matches the IP scheme for your network. Next, I'll go over each line in order to give you a deeper understanding of what each line of this configuration file is responsible for:

```
$TTL 1D
```

The **Time to Live** (**TTL**) determines how long a record may be cached within a DNS server. If you recall from earlier, where we practiced with the `dig` command, you saw that the second time you queried a domain with `dig`, the query time was less than the first time you ran the command. This is because your DNS server cached the result, but it won't hold onto it forever. At some point, the lookup will expire. The next time you look up that same domain after the cached result has expired, your server will go out and fetch the result from the DNS server again. In my examples, I used Google's DNS servers. That means at some point, your server will query those servers again once the record times out.

With the **Start of Authority** (**SOA**) line, we're establishing that our DNS server is authoritative over the `local.lan` domain:

```
@ IN SOA local.lan. hostmaster.local.lan. (
```

We also set `hostmaster@local.lan` as the email address of the responsible party for this server, but we enter it here in a different format for `bind` (`hostmaster.local.lan`). This is obviously a fake address, but for the purposes of an internal DNS server, its validity is of no concern.

Of all the lines of configuration within a zone file, `serial` is by far the one that will frustrate us the most:

```
202008161; serial
```

This is because it's not enough to simply update the zone file any time we make a change to it (change an IP address, add or remove a host, and so on); we also need to remember to increase the serial number by at least one. If we don't, `bind` won't be aware that we've made any changes, as it will look at the serial number before the rest of the file. The problem with this is that you and I are both human, and we're prone to forgetting things. I've forgotten to update `serial` many times and have become frustrated when the DNS server refused to resolve new hosts that were recently added. Once I remembered that I didn't increment the serial number, the issue was resolved after I did. Therefore, it's very important for you to remember that any time you make a change to any zone file, you'll need to also increment the serial number.

The format doesn't really matter; I used 202008161, which is simply the year, two-digit month, two-digit day, and an extra number to cover us if we make more than one change in a day (which can sometimes happen). As long as you increment the serial number by one every time you modify your zone file, you'll be in good shape—regardless of what format you use. However, the sample format I gave here is actually quite common in the field.

These values control how often secondary DNS servers will be instructed to check in for updates:

```
8H ; refresh
4H ; retry
4W ; expire
1D ) ; minimum
```

With the example refresh value, we're instructing any secondary DNS servers to check in every eight hours to see whether or not the zone records were updated. The retry field dictates how long the secondary will wait to check in, in case there was an error doing so the last time. The last two options in this section, `expire` and `minimum`, set the minimum and maximum age of the zone file, respectively. As I mentioned though, a full discussion of DNS with `bind` could constitute an entire book on its own. For now, I would just use these values until you have a reason to need to experiment. Here, we identify the name server itself:

```
IN A 192.168.1.1
@ IN NS hermes.local.lan.
```

In my case, the server is called `hermes` and it's located at `192.168.1.1`.

Next, in our file, we'll have several host entries to allow our resources to be resolved on our network by name:

```
fileserv        IN  A   192.168.1.3
hermes          IN  A    192.168.1.1
mailserv        IN  A   192.168.1.5
mail            IN  CNAME   mailserv.
web01           IN  A   192.168.1.7
```

In that example, I have three hosts: `fileserv`, `mailserv`, and `web01`. In the example, these are all address records, which means that any time our server is asked to resolve one of these names, it will respond with the corresponding IP address. If our DNS server is set as a machine's primary DNS server, it will respond with `192.168.1.3` when asked for `fileserv` and `192.168.1.7` when asked for `web01`.

The entry for `mail` is special as it is not an address record, but instead a **Canonical Name** (**CNAME**) record. In this case, it just points back to `mailserv`. Essentially, that's what a CNAME record does: it creates a pointer to another resource. In this case, if someone tries to access a server named `mail`, we redirect them to the actual server `mailserv`. Notice that on the CNAME record, we're not inputting an IP address, but instead the hostname of the resource it's linked to.

In addition, you should also notice that I added the DNS server itself (`hermes`) to the file as well. You can see it on the second line above. I've found that if you don't do this, the DNS server may complain and refuse to load the file.

Now that we have a zone file in place, we should be able to start using it. First, we'll need to restart the `bind9` service:

```
sudo systemctl restart bind9
```

After the command finishes, check to see if there are any errors:

```
systemctl status bind9
```

You should see that the service state is `active (running)`, and in addition, you should see a line telling you that the serial number for your zone file was loaded. If you see that the service is not running and/or your zone file was not loaded, you should see specific information in the output while checking the status that should point you in the right direction. If not, you can also check the system log for clues regarding `bind` as well:

```
cat /var/log/syslog | grep bind9
```

The most common mistakes I've seen typically result from not being consistent within the file. For example, if you're using a different IP scheme (such as `10.10.10.0/24`), you'll want to make sure you didn't forget to replace any of my example IP addresses with the proper scheme. Assuming that everything went smoothly, you should be able to point devices on your network to use this new DNS server. Make sure you test not only pinging devices local to your network, but outside resources as well, such as websites. If the DNS server is working properly, it should resolve your local names, and then forward your requests to your external DNS servers (the two we set as forwarders) if it doesn't find what you're looking for locally. In addition, you'll also want to make sure that port `53` is open in your network's firewall, which is the port that DNS uses. It's extremely rare that this would be an issue, but I have seen it happen.

To further test our DNS server, we can use the `dig` command, as we did before while we were experimenting with caching. Try `dig` against a local server on your LAN, as well as a DNS address that's not located on your LAN (change the first domain to an actual domain on your LAN):

```
dig webserv.local.lan
dig www.packtpub.com
```

You should see a response similar to the following:

```
;; Query time: 1 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
;; WHEN: Sat Feb 10 10:00:59 EST 2020
;; MSG SIZE  rcvd: 83
```

What you're looking for here is for both local resources and external websites to be resolvable now. You'll probably notice that the DNS server used in the output will most likely show up as a localhost address, as it did in my output, and not the address of the DNS server we just set up. Actually, you can ignore this. Most distributions of Linux nowadays use local resolvers, which essentially cache DNS lookup results on your local computer. Your computer is still using the DNS server we set up, but there's just an additional layer in between your computer and the DNS server. You can verify this with the following command:

```
systemd-resolve --status | grep "DNS Servers"
```

The output will show you the IP address of the actual server that's responding to your DNS lookups.

Next, let's take a look at the process of setting up an internet gateway, which is an option to consider if you don't already have a router or firewall on your network that acts as a device between your internet connection and your internal network.

## Setting up an internet gateway

As long as we're setting up network services, we may as well go all the way and set up a router to act as a **gateway** for our network. A gateway within a network is the device you go through to route from one network to another. In this context, the **internet gateway** will be the device that sits between your local network and the device that provides your internet connection (such as a cable modem). The gateway in a typical network is usually a commercial router or firewall, which often also provides DNS, DHCP, and routing services as well.

If you already have such a device on your network providing these services, then there's nothing for you to do. You can skip this section. But if you'd like to set up your own router, then feel free to proceed.

If you'd like to proceed and set up a router, then the first order of business is to decide which device on your network will serve that purpose. Often, administrators will build DNS, DHCP, and routing services all into the same server, so you can even use the same device you've used earlier to work through the DNS and DHCP examples for our purposes in this section. In order for a device to function as a gateway, it should have at least two network interfaces, one to your ISP device (such as a cable modem) and another interface connected to a network switch that your other servers will connect to. The interface connected to your ISP device should use DHCP, so it will obtain an IP address directly from your ISP. This interface may need a static IP with details provided from the ISP, if relevant.

Depending on what kind of internet connection you have, Linux itself can likely replace whatever device your internet modem connects to. A good example of this is a cable modem that your office or home router may utilize. In this case, the modem provides your internet connection, and then your router allows other devices on your network to access it. In some cases, your modem and router may even be the same device. Therefore, depending on the hardware you have, this method of setting up your networking may or may not be efficient. But if you do have the hardware available, you'll be able to manage the entire networking stack with Ubuntu Server quite easily.

Why might you want to create your own internet gateway? One potential reason is that it's often the case that security patches aren't provided for commercial routers and firewall devices. New vulnerabilities are discovered all the time, and if your router or firewall is no longer supported by the manufacturer, it may allow outside threats into your network. By setting up an internet gateway with Ubuntu, you'll benefit from the regular updates that Canonical provides. As long as you're using a version of Ubuntu that's still supported (such as an LTS release, as is the case with Ubuntu 20.04), you'll benefit from a more secure platform. If nothing else, a physical server that we can install Ubuntu on will usually have a more powerful CPU than a commercial device would have, which would mean that CPU bottlenecks slowing down network performance would be less likely to happen.

Thankfully, setting up an internet gateway is easy. In fact, we'll only need to execute a single command to set up routing between interfaces, which is technically all that's required in order to set up an internet gateway. But before we get into that, it's also important to keep in mind that if you do set up an internet gateway, you'll need to pay special attention to security. The device that sits between your network and your modem will be a constant attack target, just like any other gateway device would be. When it comes to commercial routers, they're also attacked constantly.

However, in most cases, they'll have some sort of default security or firewall built in. In all honesty, the security features built into common routing equipment are extremely poor and most of them are easy to hack when someone wants to badly enough. The point is that these devices have some sort of security to begin with (regardless of how good or bad), whereas a custom internet gateway of your own won't have any security at all until you add it.

When you set up an internet gateway, you'll want to pay special attention to setting up the firewall, restricting access to SSH, using very strong passwords, keeping up to date on security patches, and installing an authentication monitor such as `fail2ban`. We'll get into those topics in *Chapter 21*, *Securing Your Server*. The reason I bring this up now, though, is that if you do set up an internet gateway, you'll probably want to take a detour and read that chapter right away, just to make sure that you secure it properly.

Anyway, let's move on. A proper internet gateway, as I've mentioned, will have two Ethernet ports. On the first, you'll plug in your cable modem or internet device, and you'll connect a switch to the second. By default though, routing between these interfaces will be disabled, so traffic won't be able to move from one Ethernet port to the other. To rectify this, use the following command:

```
echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward
```

That's actually it. With that single command, you've just made your server into a router. However, that change will not survive a reboot. To make it permanent, open the `/etc/sysctl.conf` file in your editor:

```
sudo nano /etc/sysctl.conf
```

Look for the following line:

```
#net.ipv4.ip_forward=1
```

Uncomment the line by removing the hash symbol in front of it, and save the file. With that change made, your server will allow routing between interfaces even after a reboot. Of all the topics we've covered in this chapter, that one was probably the simplest. However, I must remind you again to definitely secure your server if it's your frontend device to the internet, as computer security students always enjoy practicing on a real-life Linux server. With good security practices, you'll help ensure that they'll leave you alone, or at least have a harder time breaking in.

From here, all you should need to do is attach a network switch to your other network interface, and then you can attach your other wired Ethernet devices and wireless access point to the switch. Now, Ubuntu Server is managing your entire network! Next, we will ensure the clocks of our servers are up to date by setting up NTP.

## Keeping your clock in sync with NTP

It's incredibly important for Linux servers to keep their time synchronized, as strange things can happen when a server's clock is wrong. One issue I've run into that's especially problematic is file synchronization utilities, which will exhibit strange behavior when the underlying clock of the server is incorrect. However, Ubuntu servers provide packages for the NTP client and server within the default repositories to help keep your time in sync.

Before we set up NTP, we should first ensure that our server is set up for the correct time zone. If you use the `timedatectl` command with no options, it will show you what your current settings are. If I run `timedatectl` on one of my servers, I see the following output:

```
Time zone: America/Detroit (EDT, -0400)
```

For my use case, that's the proper setting. On your end though, the results may not be correct. If not, then we can set the appropriate time zone also with the `timedatectl` command. But in order to do that, we need to know what the appropriate setting is for our area. The following command will provide us with a list:

```
timedatectl list-timezones
```

The result of that command will dump quite a few different time zones onto your screen. You can use the up and down arrows to scroll through the results. When you do find the timezone that matches your area, you can set it with a command similar to the following:

```
sudo timedatectl set-timezone America/Detroit
```

In my case, I'm setting my timezone to `America/Detroit`, which it already was. If my timezone setting was incorrect, that would've fixed it. Essentially, we just want to ensure our timezone setting is correct at this point before we continue on.

To facilitate keeping our clock synchronized, Ubuntu Server 20.04 includes the `systemd-timesyncd` package by default, which should already be present in your installation. This daemon runs in the background, and keeps your clock in sync. If it's working properly, incorrect time on your server should not be a problem. You can verify the current date and time by simply running the `date` command. You can also check the status of the `systemd-timesyncd` daemon to ensure that it's working:

```
systemctl status systemd-timesyncd
```

The status should show `active (running)`. If there are no errors, and the `date` command reports the correct time, then there should be nothing you need to do here if all you're wanting to achieve is to make sure that your clock is accurate.

If you'd like to set up your own NTP server for other nodes on your network to synchronize time against, then there's more work that you'll need to do. If you're wondering why you may want to run your own NTP server, one reason might be that if only one server on your network synchronizes with an external server (which is what NTP does), then you minimize the number of devices attempting to communicate with an external server. It also gives you more control over your network. Running your own NTP server is completely optional, but it may benefit you to do so. If this is something you'd like to set up, continue reading through this section.

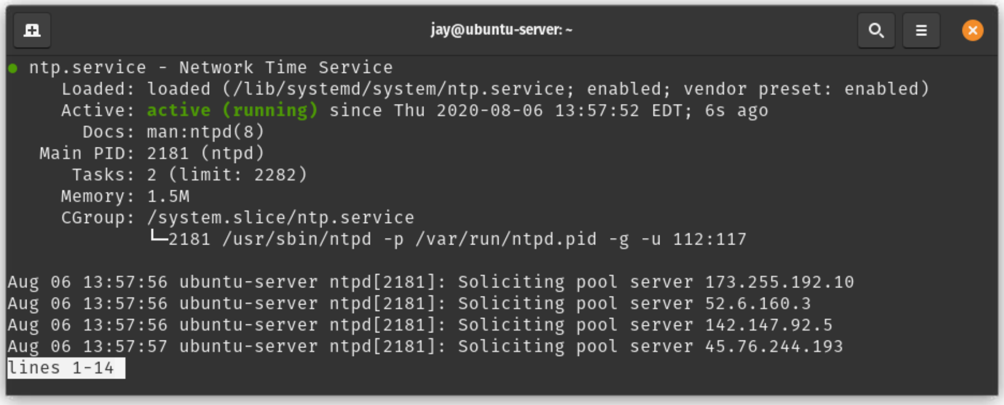
To set up an NTP server, you will need to install the `ntp` package:

```
sudo apt install ntp
```

This will actually trigger the `systemd-timesyncd` package to be removed, as `ntp` replaces it. But once installed, the `ntp` daemon will immediately start and will keep your time up to date just like `systemd-timesyncd` did. To verify, check the status of the `ntp` daemon with the following command:

```
systemctl status ntp
```

The output should show that `ntp` is running:



```
● ntp.service - Network Time Service
     Loaded: loaded (/lib/systemd/system/ntp.service; enabled; vendor preset: enabled)
     Active: active (running) since Thu 2020-08-06 13:57:52 EDT; 6s ago
       Docs: man:ntpd(8)
   Main PID: 2181 (ntpd)
      Tasks: 2 (limit: 2282)
     Memory: 1.5M
     CGroup: /system.slice/ntp.service
             └─2181 /usr/sbin/ntpd -p /var/run/ntpd.pid -g -u 112:117

Aug 06 13:57:56 ubuntu-server ntpd[2181]: Soliciting pool server 173.255.192.10
Aug 06 13:57:56 ubuntu-server ntpd[2181]: Soliciting pool server 52.6.160.3
Aug 06 13:57:56 ubuntu-server ntpd[2181]: Soliciting pool server 142.147.92.5
Aug 06 13:57:57 ubuntu-server ntpd[2181]: Soliciting pool server 45.76.244.193
lines 1-14
```

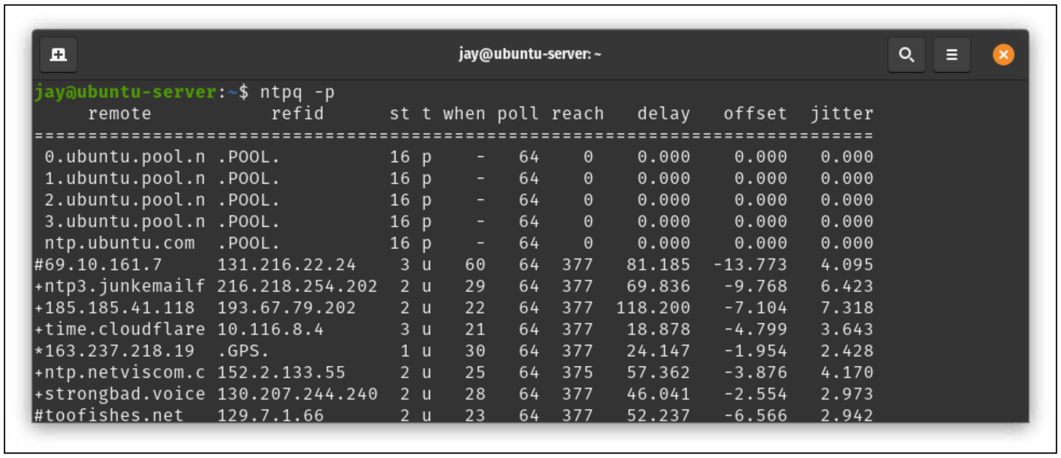
Figure 10.4: Checking the status of the NTP service

You'll find the configuration file for this newly installed daemon at `/etc/ntp.conf`, which will contain some lines of configuration detailing which servers your local `ntp` daemon will attempt to synchronize time with:

```
pool 0.ubuntu.pool.ntp.org iburst
pool 1.ubuntu.pool.ntp.org iburst
pool 2.ubuntu.pool.ntp.org iburst
pool 3.ubuntu.pool.ntp.org iburst
```

As you can see, our server will synchronize with Ubuntu's time servers if we leave the default configuration as it is. For most users, that's perfectly fine. There's nothing wrong with using Ubuntu's servers. If you'd like to have your server synchronize with other time servers, you can set that here. At this point, we actually have an NTP server set up. Believe it or not, it was that easy. When we installed the `ntp` package, the server began using it to synchronize to Ubuntu's time servers. Now that the server is synchronized, you can have other devices on your network synchronize to it.

Before we get ahead of ourselves though, we should make sure that the server we installed NTP on is synchronizing properly. This will give us a chance to use the `ntpq` command, which we can use to view statistics about how well our server is synchronizing. The `ntpq -p` command should print out statistics we can use to verify connectivity:



```
jay@ubuntu-server:~$ ntpq -p
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
 0.ubuntu.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.000
 1.ubuntu.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.000
 2.ubuntu.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.000
 3.ubuntu.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.000
 ntp.ubuntu.com  .POOL.          16 p    -   64    0    0.000    0.000   0.000
#69.10.161.7     131.216.22.24    3 u   60   64  377   81.185  -13.773   4.095
+ntp3.junkemailf 216.218.254.202  2 u   29   64  377   69.836   -9.768   6.423
+185.185.41.118  193.67.79.202    2 u   22   64  377  118.200   -7.104   7.318
+time.cloudflare 10.116.8.4       3 u   21   64  377   18.878   -4.799   3.643
*163.237.218.19  .GPS.            1 u   30   64  377   24.147   -1.954   2.428
+ntp.netviscom.c 152.2.133.55     2 u   25   64  375   57.362   -3.876   4.170
+strongbad.voice 130.207.244.240  2 u   28   64  377   46.041   -2.554   2.973
#toofishes.net   129.7.1.66       2 u   23   64  377   52.237   -6.566   2.942
```

Figure 10.5: Example output from the ntpq -p command

To better understand the output of the `ntpq -p` command, I'll go through each column. First, the `remote` column details the NTP servers we're connected to. The `refid` column refers to the NTP servers that the remote servers are connected to. The `st` column refers to a server's stratum, which refers to how close the server is from us (the lower the number, the closer, and typically, better). The `t` column refers to the type of connection, which it shows as a single letter, such as `u`, `b`, or `A`, which correspond to **unicast**, **broadcast**, or **manycast** respectively. You can view more information about the various types of connections by perusing the man page for `ntpq`:

```
man ntpq
```

Continuing, the `when` column refers to how long it has been since the last time the server was polled. The `poll` column indicates how often the server will be polled, which is 64 seconds for most of the entries in the example screenshot. The `reach` column contains the results of the most recent eight NTP updates. If all eight are successful, this field will read `377`. When you first start the `ntp` daemon on your server, it may take some time for this number to reach `377`. If you're not able to get a value of `377` for this field, one possibility is that your server is not able to reach external NTP servers for synchronization.

Finally, we have the `delay`, `offset`, and `jitter` columns. The `offset` column refers to the delay in reaching the server, in milliseconds. `offset` references the difference between the local clock and the server's clock. Finally, the `jitter` column refers to the network latency between your server and theirs.

Assuming that the results of the `ntpq -p` command look good and your server is synchronizing properly, you already essentially have an NTP server at your disposal, since there is little difference between a client and a server. As long as your server is synchronizing properly and port `123` is open between them through the firewall, you should be able to reconfigure your nodes to connect to your server, by changing the pool addresses in your client's configuration (within `/etc/ntp.conf`) to point to either the IP address or the **Fully Qualified Domain Name** (**FQDN**) of your NTP server instead of Ubuntu's servers.

Before we close out this chapter, there is one additional change you should consider implementing to the `/etc/ntp.conf` file. Look for the following line within that file:

```
#restrict 192.168.123.0 mask 255.255.255.0 notrust
```

Change this line by first uncommenting it, changing the network address and subnet mask to match the details for your network, and then remove the `notrust` keyword at the end. The line should look similar to the following, depending on your network configuration:

```
restrict 192.168.1.0 mask 255.255.255.0
```

So, what does this option do for us? Basically, we're limiting access to our NTP server to local clients only, and we're only allowing read-only access for security purposes.

Now that you have a working NTP server, feel free to experiment and point your existing nodes to it and have them synchronize. To have devices on your network start using your new NTP server, it's as simple as configuring your DHCP server to point your devices to the IP address or DNS name of your NTP server as the designated time server for your network, and they'll start synchronizing with it as soon as their DHCP leases expire and renew. If you're utilizing the `isc-dhcp-server` service we set up earlier for DHCP in your network, we can add the relevant option (shown in bold) to our `/etc/dhcp/dhcpd.conf` file:

```
subnet 192.168.1.0 netmask 255.255.255.0 {
  range 192.168.1.100 192.168.1.240;
  option routers 192.168.1.1;
  option domain-name-servers 192.168.1.1;
  option ntp-servers 192.168.1.1;
}
```

Depending on the size of your network, a local NTP server may or may not make sense, but at the very least, NTP should be installed on every Linux workstation and server to ensure proper time synchronization. In most cases, the workstation version of Ubuntu will already be configured to synchronize to the Ubuntu time servers, but when it comes to the server version, NTP isn't typically installed for you.

## Summary

In this chapter, we explored additional networking topics. We started off with some notes on planning an IP address scheme for your network so that you could create groups for the different types of nodes, such as servers and network equipment, and plan a pool of addresses for DHCP. We also worked through the process of setting up a DHCP and DNS server, which gives us additional flexibility when configuring the services we run on our network, such as when defining a custom IP scheme for DHCP, as well as giving us the ability to resolve the hostnames of devices on our network by name. We closed off this chapter with discussions on setting up an internet gateway to serve as our internet-facing router, as well as configuring an NTP server for ensuring that the clocks of our servers are synchronized.

In the next chapter, we'll take a look at sharing and transferring files over the network. This will include covering NFS and Samba shares, as well as using `scp`, `rsync`, and `sshfs`. Stay tuned!

## Further reading

- 8 Steps to Understanding IP Subnetting: `https://www.techopedia.com/6/28587/internet/8-steps-to-understanding-ip-subnetting`
- IP Subnet Calculator: `https://www.calculator.net/ip-subnet-calculator.html`

# 12
# Sharing and Transferring Files

In the previous chapter, we looked at what's involved in the process of setting up a few network services, such as DHCP and DNS. Those are two important components of a network, but there are quite a few different types of applications and resources you can make available on your network to further enhance it. A file server is one such example, which can give your users a central place to store critical files and can even enhance collaboration.

Perhaps you've used a file server before, or even set one up on a different platform. With Ubuntu Server, there are multiple methods to not only store files, but also to transfer files from one node to another over a network link. In this chapter, we'll look into setting up a central file server using both Samba and NFS, as well as how to transfer files between nodes with utilities such as `scp` and `rsync`. We'll also go over some situations in which one solution may work better for you than another. As we go through these concepts, we will cover the following topics:

- File server considerations
- Sharing files with Windows users via Samba
- Setting up NFS shares
- Transferring files with `rsync`
- Transferring files with `scp`
- Mounting remote directories with SSHFS

Before we can get started with configuring our server to enable it to share files with other users, we should first understand what our available options are to enable us to choose the best technology for our use case.

## File server considerations

When it comes to setting up a file server, the process is a matter of setting up some sort of daemon to accept connections and share specific directories, and ensure the appropriate users are able to access those directories. You'll also implement permissions to determine who can access specific directories, and what type of access they will have (read/write, read-only, and so on). When deciding *how* to share the files, it's generally a choice between two common technologies that can facilitate the actual sharing, **Samba** and **NFS**.

All in all, there's nothing stopping you from hosting both Samba and NFS shares on a single server. The two technologies can actually co-exist on the same device. However, each of the two popular solutions is valid for particular use cases. Before we get started with setting up our file server, we should first understand the differences between Samba and NFS, so we can make an informed decision as to which one is more appropriate for our environment. As a general rule of thumb, Samba is great for mixed environments (where you have Windows as well as Linux clients), and NFS is more appropriate for use in Linux or Unix environments, but there's a bit more to it than that.

Samba is a great solution for many environments, because it allows you to share files with Windows, Linux, and macOS machines. Basically, pretty much everyone will be able to access your shares, provided you give them permission to do so. The reason this works is because Samba is a re-implementation of the **Server Message Block** (**SMB**) protocol, which is primarily used by Windows systems. However, you don't need to use the Windows platform in order to be able to access Samba shares, since many platforms offer support for this protocol. Even Android phones are able to access Samba file shares with the appropriate app, as well as other platforms.

You may be wondering why I am going to cover two different solutions in this chapter. After all, if Samba shares can be accessed by pretty much everything and everyone, why bother with anything else? Even with Samba's many strengths, there are also weaknesses as well. First of all, permissions are handled very differently, so you'll need to configure your shares in specific ways in order to prevent access to users that shouldn't be able to see confidential data. With NFS, full support for standard UNIX permissions is provided, so you'll only need to configure your permissions once. If permissions and confidentiality are important to you, you may want to look closer at NFS.

It's also not accurate to say that Windows systems cannot access NFS shares, because some versions actually can. By default, no version of Windows supports NFS outright, but some editions offer a plugin you can install that enables this support. The name of the NFS plugin in Windows has changed from one version to another (such as **Services for UNIX**, **Subsystem for UNIX-based Applications**, **NFS Client**, and most recently, **Windows Subsystem for Linux**) but the idea is the same. In the past, Microsoft required a more expensive Windows license on your laptop or desktop to allow the installation of the NFS client, but that's no longer the case. The Windows Subsystem for Linux can be installed on any version of Windows 10, so this licensing restriction only comes into play on older versions. If you do have legacy Windows machines in use (which is becoming increasingly rare), using NFS may actually increase costs. In that situation, Samba is a clear winner.

Regarding an all-Linux environment or a situation where you only have Linux machines that need to access your shares, NFS is a great choice because it integrates much more tightly with the rest of the distribution. Permissions can be more easily enforced and, depending on your hardware, performance may be higher. The specifics of your computing environment will ultimately make your decision for you. Perhaps you'll choose Samba for your mixed environment, or NFS for your all-Linux environment. Maybe you'll even set up both NFS and Samba, having shares available for each platform. My recommendation is to learn and practice both, since you'll use both solutions at one point or another during your career anyway.

Before you continue to the sections on setting up Samba and NFS, I recommend you first decide where in your filesystem you'd like to act as a parent directory for your file shares. This isn't actually required, but I think it makes for better organization. There is no one right place to store your shares, but personally I like to create a `/share` directory at the `root` filesystem and create sub-directories for my network shares within it. For example, I can create `/share/documents`, `/share/public`, and so on for Samba shares. With regard to NFS, I usually create shared directories within `/exports`. You can choose how to set up your directory structure. As you read the remainder of this chapter, make sure to change my example paths to match yours if you use a different style.

## Sharing files with Windows users via Samba

In this section, I'll walk you through setting up your very own **Samba** file server. I'll also go over a sample configuration to get you started so that you can add your own shares. One feature that Samba supports is integration with Active Directory, but that's outside the scope of this book as that's a feature specific to Windows Server.

I mention that here because our Samba implementation will be relatively wide open, which is a bad practice. With Active Directory, you can have more effective control over user access. But to keep it simple, we'll create a simple Samba server to get you started, and then from there, you can research more complex implementations if it makes sense to do so for your organization.

First, we'll need to make sure that the `samba` package is installed on our server:

```
sudo apt install samba
```

When you install the `samba` package, you'll have a new daemon installed on your server, `smbd`. The `smbd` daemon will be automatically started and enabled for you. You'll also be provided with a default configuration file for Samba, located at `/etc/samba/smb.conf`. For now, I recommend stopping `samba` since we have yet to configure it:

```
sudo systemctl stop smbd
```

Since we're going to configure Samba from scratch, we should start with an empty configuration file. Let's back up the original file, rather than overwrite it. The default file includes some useful notes and samples, so we should keep it around for future reference:

```
sudo mv /etc/samba/smb.conf /etc/samba/smb.conf.orig
```

Now, we can begin a fresh configuration. Although it's not required, I like to split my Samba configuration up between two files, `/etc/samba/smb.conf` and `/etc/samba/smbshared.conf`. You don't have to do this, but I think it makes the configuration cleaner and easier to read. First, here is a sample `/etc/samba/smb.conf` file:

```
[global]
server string = File Server
workgroup = WORKGROUP
security = user
map to guest = Bad User
name resolve order = bcast wins
include = /etc/samba/smbshared.conf
```

As you can see, this is a really short file. Basically, we're including only the lines we absolutely need to in order to set up a file server with Samba. Next, I'll explain each line and what it does.

```
[global]
```

With the [global] stanza, we're declaring the global section of our configuration, which will consist of settings that will impact Samba as a whole. There will also be additional stanzas for individual shares, which we'll get to later.

```
server string = File Server
```

The server string is somewhat of a description field for the File Server. If you've browsed networks from Windows computers before, you may have seen this field. Whatever you type here will display underneath the server's name in Windows Explorer. This isn't required, but it's nice to have.

```
workgroup = WORKGROUP
```

Here, we're setting the workgroup, which is the exact same thing as a workgroup on Windows PCs. In short, the workgroup is a namespace that describes a group of machines. When browsing network shares on Windows systems, you'll see a list of workgroups, and then one or more computers within that workgroup. In short, this is a way to logically group your nodes. You can set this to whatever you like. If you already have a workgroup in your organization, you should set it here to match the workgroup names of your other machines. The default workgroup name is simply WORKGROUP on Windows PCs, if you haven't customized the workgroup name at all.

```
security = user
```

This setting sets up Samba to utilize usernames and passwords for authentication to the server. Here, we're setting the security mode to user, which means we're using local users to authenticate, rather than other options such as ads (Active Directory) or domain (domain controller), which are both outside the scope of this book.

```
map to guest = Bad User
```

This option configures Samba to treat unauthenticated users as guest users. Basically, unauthenticated users will still be able to access shares, but they will have guest permissions instead of full permissions. If that's not something you want, then you can omit this line from your file. Note that if you do omit this, you'll need to make sure that both your server and client PCs have the same user account names on either side. Ideally, we want to use directory-based authentication, but that's beyond the scope of this book.

```
name resolve order = bcast wins
```

The `name resolve order` setting configures how Samba resolves hostnames. In this case, we're using the broadcast name first, followed by `wins`. Since `wins` has been pretty much abandoned (and replaced by DNS), we include it here solely for compatibility with legacy networks.

```
include = /etc/samba/smbshared.conf
```

Remember how I mentioned that I usually split my Samba configurations into two different files? On this line, I'm calling that second `/etc/samba/smbshared.conf` file. The contents of the `smbshared.conf` file will be inserted right here, as if we only had one file. We haven't created the `smbshared.conf` file yet. Let's take care of that next. Here's a sample `smbshared.conf` file:

```
[Documents]
path = /share/documents
force user = myuser
force group = users
public = yes
writable = no

[Public]
path = /share/public
force user = myuser
force group = users
create mask = 0664
force create mode = 0664
directory mask = 0777
force directory mode = 0777
public = yes
writable = yes
```

As you can see, I'm separating share declarations in their own file. We can see several interesting things within `smbshared.conf`. First, we have two stanzas, `[Documents]` and `[Public]`. Each stanza is a share name, which will allow Windows users to access the share under `//servername/share-name`. In this case, this file will give us two shares: `//servername/Documents` and `//servername/Public`. The `Public` share is writable for everyone, though the `Documents` share is restricted to read only. The `Documents` share has the following options:

```
path = /share/documents
```

This is the path to the share, which must exist on the server's filesystem. In this case, when a user reads files from `//servername/Documents` on a Windows system, they will be reading data from `/share/documents` on the Ubuntu server that's housing the share.

```
force user = myuser
force group = users
```

These two lines are basically bypassing user ownership. When a user reads this share, they are treated as `myuser` instead of their actual user account. Normally, you would want to set up LDAP or Active Directory to manage your user accounts and handle their mapping to Ubuntu Server, but a full discussion of directory-based user access is beyond the scope of this book, so I provided the `force` options as an easy starting point. The user account you set here must exist on the server.

```
public = yes
writable = no
```

With these two lines, we're configuring what users are able to do once they connect to this share. In this case, `public = yes` means that the share is publicly available, though `writable = no` prevents anyone from making changes to the contents of this share. This is useful if you want to share files with others, but you want to restrict access and stop anyone from being able to modify the content.

The `Public` share has some additional settings that weren't found in the `Documents` share:

```
create mask = 0664
force create mode = 0664
directory mask = 0777
force directory mode = 0777
```

With these options, I'm setting up how the permissions of files and directories will be handled when new content is added to the share. Directories will be given `777` permissions and files will be given permissions of `664`. Yes, these permissions are very open; note that the share is named `Public`, which implies full access anyway, and its intended purpose is to house data that isn't confidential or restricted:

```
public = yes
writable = yes
```

Just as I did with the previous share, I'm setting up the share to be publicly available, but this time I'm also configuring it to allow users to make changes.

To take advantage of this configuration, we need to start the Samba daemon. Before we do though, we want to double-check that the directories we entered into our `smbshared.conf` file exist, so if you're using my example, you'll need to create `/share/documents` and `/share/public`:

```
sudo mkdir /share
sudo mkdir /share/documents
sudo mkdir /share/public
```

Also, the user account that was referenced in the force user and the group referenced in the force group must both exist and have ownership over the shared directories:

```
sudo chown -R jay /share
```

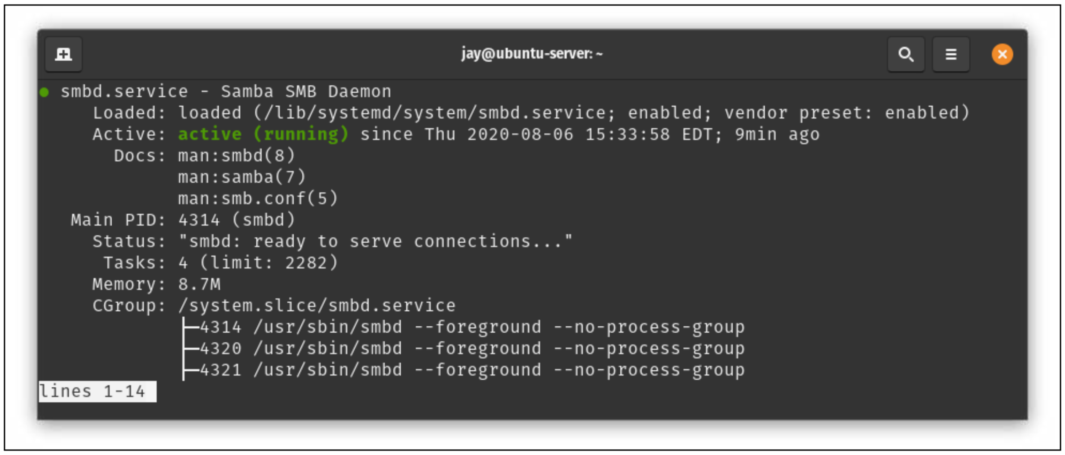
At this point, it's a good idea to use the `testparm` command, which will test the syntax of our Samba configuration files for us. It won't necessarily catch every error we could have made, but it is a good command to run to quickly check the sanity. This command will first check the syntax, then it will print the entire configuration to the terminal for you to have a chance to review it. If you see no errors here, then you can proceed to start the service:

```
sudo systemctl start smbd
```

Then, check the status to ensure that the daemon is running:

```
sudo systemctl status smbd
```

This will produce output similar to the following:



```
● smbd.service - Samba SMB Daemon
     Loaded: loaded (/lib/systemd/system/smbd.service; enabled; vendor preset: enabled)
     Active: active (running) since Thu 2020-08-06 15:33:58 EDT; 9min ago
       Docs: man:smbd(8)
             man:samba(7)
             man:smb.conf(5)
   Main PID: 4314 (smbd)
     Status: "smbd: ready to serve connections..."
      Tasks: 4 (limit: 2282)
     Memory: 8.7M
     CGroup: /system.slice/smbd.service
             ├─4314 /usr/sbin/smbd --foreground --no-process-group
             ├─4320 /usr/sbin/smbd --foreground --no-process-group
             ├─4321 /usr/sbin/smbd --foreground --no-process-group
lines 1-14
```

Figure 12.1: Checking the status of the smbd daemon

That really should be all there is to it; you should now have `Documents` and `Public` shares on your file server that Windows users should be able to access. In fact, your Linux machines should be able to access these shares as well. On Windows, **Windows Explorer** has the ability to browse file shares on your network. If in doubt, try pressing the *Windows* key and the *r* key at the same time to open the **Run** dialog box, and then type the **Universal Naming Convention** (**UNC**) path to the share (`\\servername\Documents` or `\\servername\Public`). You should be able to see any files stored in either of those directories. In the case of the `Public` share, you should be able to create new files there as well.

On Linux systems, if you have a desktop environment installed, most of them feature a file manager that supports browsing network shares. Since there are a handful of different desktop environments available, the method varies from one distribution or configuration to another. Typically, most Linux file managers will have a network link within the file manager, which will allow you to easily browse your local shares:

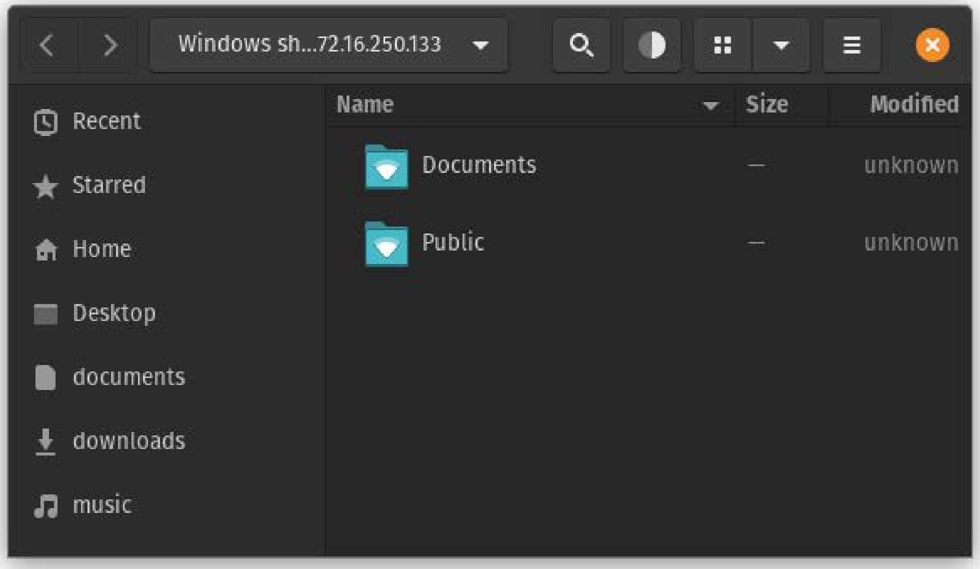


Figure 12.2: Browsing a Samba share from a Linux client

If your file manager doesn't show you the available shares on your server, you can also access a Samba share by adding an entry for it in the `/etc/fstab` file, such as the following:

```
//myserver/share/documents /mnt/documents cifs username=myuser,noauto 0
0
```

In order for the `fstab` entry to work, your Linux client will need to have the Samba client packages installed. If your distribution is Debian-based (such as Ubuntu), you will need to install the `smbclient` and `cifs-utils` packages:

```
sudo apt install smbclient cifs-utils
```

Then, assuming the local directory exists (`/mnt/documents` in the example `fstab` line), you should be able to mount the share with the following command:

```
sudo mount /mnt/documents
```

In the `fstab` entry, I included the `noauto` option so that your system won't mount the Samba share at boot time (you'll need to do so manually with the `mount` command). If you do want the Samba share automatically mounted at boot time, change `noauto` to `auto`. However, you may receive errors during boot if for some reason the server hosting your Samba shares isn't accessible, which is why I prefer the `noauto` option.

If you'd prefer to mount the Samba share without adding an `fstab` entry, the following example command should do the trick; just change the share name and mount point to match your local configuration:

```
sudo mount -t cifs //myserver/Documents -o username=myuser /mnt/
documents
```

From here, feel free to experiment. You can add additional shares as appropriate, and customize your Samba implementation as you see fit. In the next section, we'll explore NFS.

## Setting up NFS shares

An alternative to Samba is **Network File System** (**NFS**). It's a great method of sharing files from a Linux or Unix server to a Linux or Unix server. As I mentioned earlier in the chapter, Windows systems can access NFS shares as well, but that requires an add-on to be enabled. Therefore, NFS is preferred in a Linux or Unix environment, since it fully supports Linux- and Unix-style permissions. As you can see from our dive into Samba earlier, we essentially forced all shares to be treated as being accessed by a particular user, which was messy, but was the easiest example of setting up a Samba server without also walking you through setting up a Windows Active Directory controller. Samba can certainly support per-user access restrictions and benefits greatly from a centralized directory server, though that would basically be a book of its own! NFS is a bit more involved to set up, but in the long run, I think it's easier and integrates better in a non-mixed environment.

Earlier, we set up a parent directory in our filesystem to house our Samba shares, and we should do the same thing with NFS. While it wasn't mandatory to have a special parent directory with Samba (I had you do that in order to be neat, but you weren't required to), NFS really does want its own directory to house all of its shares. It's not strictly required with NFS either, but there's an added benefit in doing so, which I'll go over before the end of this section. In my case, I'll use `/exports` as an example, so you should make sure that that directory, or whatever you've chosen for NFS, exists:

```
sudo mkdir /exports
```

Next, let's install the required NFS packages on our server. The following command will install NFS and its dependencies:

```
sudo apt install nfs-kernel-server
```

Once you install the `nfs-kernel-server` package, the `nfs-kernel-server` daemon will start up automatically. It will also create a default `/etc/exports` file (which is the main file that NFS reads its share information from), but it doesn't contain any useful settings, just some commented lines. Let's back up the `/etc/exports` file, since we'll be creating our own:

```
sudo mv /etc/exports /etc/exports.orig
```

To set up NFS, let's first create some directories that we will share with other users. Each share in NFS is known as an **export**. I'll use the following directories as examples, but you can export any directory you like:

```
/exports/backup
/exports/documents
/exports/public
```

In the `/etc/exports` file (which we're creating fresh), I'll insert the following four lines:

```
/exports *(ro,fsid=0,no_subtree_check)
/exports/backup 192.168.1.0/255.255.255.0(rw,no_subtree_check)
/exports/documents 192.168.1.0/255.255.255.0(ro,no_subtree_check)
/exports/public 192.168.1.0/255.255.255.0(rw,no_subtree_check)
```

The first line is **export root**, which I'll go over a bit later. The next three lines are individual shares or exports. The `/backup`, `/documents`, and `/public` directories are being shared from the `/exports` parent directory. Each of these lines is not only specifying which directory is being shared with each export, but also which network is able to access them.

In this case, after the directory is called out in a line, we're also setting which network is able to access them (`192.168.1.0/255.255.255.0` in our case). This means that if you're connecting from a different network, your access will be denied. Each connecting machine must be a member of the `192.168.1.0/24` network in order to proceed (so make sure you change this to match your IP scheme). Finally, we include some options for each export, for example, `rw,no_subtree_check`.

As far as what these options do, the first (`rw`) is rather self-explanatory. Here, we can set whether or not other nodes will be able to make changes to data within the export. In the examples I gave, the `/documents` export is read-only (`ro`), while the others allow read and write.

The next option in each example is `no_subtree_check`. This option is known to increase reliability and is mainly implied by default. However, not including it may make NFS complain when it restarts, but nothing that will actually stop it from working. Particularly, this option disables what is known as **subtree checking**, which has had some stability issues in the past. Normally, when a directory is exported, NFS might scan parent directories as well, which is sometimes problematic, and can cause issues when it comes to open file handles.

There are several other options that can be included in an `export`, and you can read more about them by checking the man page for `export`:

```
man exports
```

One option you'll see quite often in the wild is `no_root_squash`. Normally, the `root` user on one system is mapped to nobody on the other for security reasons. In most cases, one system having `root` access to another is a bad idea. The `no_root_squash` option disables this, and it allows the `root` user on one end to be treated as the `root` user on the other. I can't think of a reason, personally, where this would be useful (or even recommended), but I have seen this option quite often in the wild, so I figured I would bring it up. Again, check the man pages for `export` for more information on additional options you can pass to your exports.

Next, we have one more file to edit before we can actually seal the deal on our NFS setup. The `/etc/idmapd.conf` file is necessary for mapping permissions on one node to another. In *Chapter 2*, *Managing User Permissions*, we talked about the fact that each user has an ID (UID) assigned to them. The problem, though, is that from one system to another, a user will not typically have the same UID. For example, user `jdoe` may be UID `1001` on server A, but `1007` on server B. When it comes to NFS, this greatly confuses the situation, because UIDs are used in order to reference permissions. Mapping IDs with `idmapd` allows this to stay consistent and handles translating each user properly, though it must be configured correctly and consistently on each node. Basically, as long as you use the same domain name on each server and client and configure the `/etc/idmapd.conf` file properly on each, you should be fine.

To configure this, open /etc/idmapd.conf in your text editor. Look for an option that is similar to the following:

```
# Domain = localdomain
```

First, remove the # symbol from that line to uncomment it. Then, change the domain to match the one used within the rest of your network. You can leave this as it is as long as it's the same on each node, but if you recall from *Chapter 11*, *Setting Up Network Services*, we used a sample domain of local.lan in our DHCP configuration, so it's best to make sure you use the same domain name everywhere—even the domain provided by DHCP. Basically, just be as consistent as you can and you'll have a much easier time overall. You'll also want to edit the /etc/idmapd.conf file on each node that will access your file server, to ensure they are configured the same as well.

With our /etc/exports and /etc/idmapd.conf files in place, and assuming you've already created the exported directories on your filesystem, we should be all set to restart NFS to activate our configuration:

```
sudo systemctl restart nfs-kernel-server
```

After restarting NFS, you should check the daemon's output via systemctl to ensure that there are no errors:

```
systemctl status nfs-kernel-server
```

As long as there are no errors, our NFS server should be working. Now, we just need to learn how to mount these shares on another system. Unlike Samba, using a Linux file manager and browsing the network will not show NFS exports by default; we'll need to mount them manually. Client machines, assuming they are Debian-based (Ubuntu fits this description) will need the nfs-common package installed in order to access these exports. It may already be installed, but if it's not, we can install it with apt like any other package:

```
sudo apt install nfs-common
```

With the client installed, we can now use the mount command to mount NFS exports on a client. For example, with regards to our documents export, we can use the following variation of the mount command to do the trick:

```
sudo mount myserver:/documents /mnt/documents
```

Replace myserver with either your server's hostname or its IP address, documents with the name of the actual share on the server, and /mnt/documents with the path on your local server where you want to mount the share. From this point forward, you should be able to access the contents of the documents export on your file server. Notice, however, that the exported directory on the server was /exports/documents, but we only asked for /documents instead of the full path with the example mount command. The reason this works is because we identified an export root of /exports. To save you from flipping back, here's the first line from the /etc/exports file, where we identified our export root:

```
/exports *(ro,fsid=0,no_subtree_check)
```

With the export root, we basically set the base directory for our NFS exports. We set it as read-only (ro), because we don't want anyone making any changes to the /exports directory itself. Other directories within /exports have their own permissions and will thus override the ro setting on a per-export basis, so there's no real reason to set our export root as anything other than read-only. With our export root set, we don't have to call out the entire path of the export when we mount it; we only need the directory name. This is why we can mount an NFS export from myserver:/documents instead of having to type the entire path. While this does save us a bit of typing, it's also useful because from the user's perspective, they aren't required to know anything about the underlying filesystem on the server. There's simply no value in the user having to memorize the fact that the server is sharing a document's directory from /exports; all they're interested in is getting to their data. Another benefit is if we ever need to move our export root to a different directory (during a maintenance period), our users won't have to change their configuration to reference the new place; they'll only need to unmount and remount the exports.

So, at this point, you'll have three directories being exported from your file server, and you can always add others as you go. However, whenever you add a new export, it won't be automatically added and read by NFS. You can restart NFS to activate new exports, but that's not really a good idea while users may be connected to them, since that will disrupt their access. Thankfully, the following command will cause NFS to reread the /etc/exports file without disrupting existing connections. This will allow you to activate new exports immediately without having to wait for users to finish what they're working on:

```
sudo exportfs -a
```

With this section out of the way, you should be able to export a directory from your Ubuntu Server, and then mount that export on another Linux machine. Feel free to practice creating and mounting exports until you get the hang of it. In addition, you should familiarize yourself with a few additional options and settings that are allowable in the /etc/exports file, after consulting the man page on exports.

When you've had more NFS practice than you can tolerate, we'll move on to a few ways in which you can copy files from one node to another without needing to set up an intermediary service or daemon.

## Transferring files with rsync

Of all the countless tools and utilities available in the Linux and Unix world, few are as beloved as `rsync`. `rsync` is a utility that you can use to copy data from one place to another very easily, and there are many options available to allow you to be very specific about how you want the data to be transferred. Examples of its many use cases include copying files while preserving permissions, copying files while backing up replaced files, and even setting up incremental backups. If you don't already know how to use `rsync`, you'll probably want to get lots of practice with it, as it's something you'll soon see will be indispensable during your career as a Linux administrator, and it is also something that the Linux community generally assumes you already know. `rsync` is not hard to learn. Most administrators can learn the basic usage in about an hour or less, but the countless options available will lead you to learn new tricks even years down the road.

Another aspect that makes `rsync` flexible is the many ways you can manipulate the source and target directories. I mentioned earlier that `rsync` is a tool you can use to copy data from one place to another. The beauty of this is that the source and target can literally be anywhere you'd like. For example, the most common usage of `rsync` is to copy data from a directory on one server to a directory on another server over the network. However, you don't even have to use the network; you can even copy data from one directory to another on the same server. While this may not seem like a useful thing to do at first, consider that the target directory may be a mount point that leads to a backup disk, or an NFS share that actually exists on another server. This also works in reverse: you can copy data from a network location to a local directory if you desire.

To get started with practicing with `rsync`, I recommend that you find some sample files to work with. Perhaps you have a collection of documents you can use, MP3 files, videos, text files, basically any kind of data you have lying around. It's important to make a copy of this data. If we make a mistake we could overwrite things, so it's best to work with a copy of the data, or data you don't care about, while you're practicing. If you don't have any files to work with, you can create some text files. The idea is to practice copying files from one place to another; it really doesn't matter what you copy or where you send it to. I'll walk you through some `rsync` examples that will progressively increase in complexity. The first few examples will show you how to back up a `home` directory, but later examples will be potentially destructive so you will probably want to work with sample files until you get the hang of it.

Here's our first example:

```
sudo rsync -r /home/myuser /backup
```

With that command, we're using `rsync` (as `root`) to copy the contents of the `home` directory for the `myuser` directory to a backup directory, `/backup` (make sure the target directory exists). In the example, I used the `-r` option, which means `rsync` will grab directories recursively as well. You should now see a copy of the `/home/myuser` directory inside your `/backup` directory.

However, we have a bit of a problem. If you look at the permissions in the `/backup/myuser` directory, you can see that everything in the target is now owned by `root`. This isn't a good thing; when you back up a user's `home` directory, you'll want to retain their permissions. In addition, you should retain as much metadata as you can, including things like timestamps. Let's try another variation of `rsync`. Don't worry about the fact that `/backup` already has a copy of the `myuser` home directory from our previous backup. Let's perform the backup again, but this time, we'll use the `-a` option:

```
sudo rsync -a /home/myuser /backup
```

This time, we replaced the `-r` option with `-a` (archive), which retains as much metadata as possible (in most cases, it should make everything an exact copy). What you should notice now is that the permissions within the backup match the permissions within the user's `home` directory we copied from. The timestamps of the files will now match as well. This works because whenever `rsync` runs, it will copy what's different from the last time it ran. The files from our first backup were already there, but the permissions were wrong. When we ran the second command, `rsync` only needed to copy what was different, so it applied the correct permissions to the files. If any new files were added to the source directory since we last ran the command, the new or updated files would be copied over as well.

The `archive` mode (the `-a` option that we used with the previous command) is actually very popular; you'll probably see it a lot during your travels. The `-a` option is actually a wrapper option that includes the following options all at the same time:

```
-rlptgoD
```

If you're curious about what each of these options do, consult the man page for `rsync` for more detailed information. In summary, the `-r` option copies data recursively (which we already know), the `-l` option copies symbolic links, `-p` preserves permissions, `-g` preserves group ownership, `-o` preserves the owner, and `-D` preserves device files. If you put those options together, we get `-rlptgoD`. Therefore, `-a` is actually equal to `-rlptgoD`. I find `-a` easier to remember.

The archive mode is great and all, but wouldn't it be nice to be able to watch what rsync is up to when it runs? Add the -v option and try the command again:

```
sudo rsync -av /home/myuser /backup
```

This time, rsync will display on your terminal what it's doing as it runs (-v activates **verbose** mode). This is actually one of my favorite variations of the rsync command, as I like to copy everything and retain all the metadata, as well as watch what rsync is doing as it works.

What if I told you that rsync supports SSH by default? It's true! Using rsync, you can easily copy data from one node to another, even over SSH. The same options apply, so you don't actually have to do anything different other than point rsync to the other server, rather than to another directory on your local server:

```
sudo rsync -av /home/myuser admin@192.168.1.5:/backup
```

With this example, I'm copying the home directory for myuser to the /backup directory on server 192.168.1.5. I'm connecting to the other server as the admin user. Make sure you change the user account and IP address accordingly, and also make sure the user account you use has access to the /backup directory. When you run this command, you should get prompted for the SSH password as you would when using plain SSH to connect to the server. After the connection is established, the files will be copied to the target server and directory.

Now, we'll get into some even cooler examples (some of which are potentially destructive), and we probably won't want to work with an actual home directory for these, unless it's a test account and you don't care about its contents. As I've mentioned before, you should have some test files to play with. When practicing, simply replace my directories with yours. Here's another variation worth trying:

```
sudo rsync -av --delete /src /target
```

Now I'm introducing you to the --delete option. This option allows you to synchronize two directories. Let me explain why this is important. With every rsync example up until now, we've been copying files from point A to point B, but we weren't deleting anything. For example, let's say you've already used rsync to copy contents from point A to point B. Then, you delete some files from point A. When you use rsync to copy files from point A to point B again, the files you deleted in point A won't be deleted in point B. They'll still be there. This is because by default, rsync copies data between two locations, but it doesn't remove anything. With the --delete option, you're effectively synchronizing the two points, thus you're telling rsync to make them the same by allowing it to delete files in the target that are no longer in the source.

Next, we'll add the `-b` (backup) option:

```
sudo rsync -avb --delete /src /target
```

This one is particularly useful. Normally, when a file is updated on `/src` and then copied over to `/target`, the copy on `/target` is overwritten with the new version. But what if you don't want any files to be replaced? The `-b` option renames files on the target that are being overwritten, so you'll still have the original file. If you add the `--backup-dir` option, things get really interesting:

```
sudo rsync -avb --delete --backup-dir=/backup/incremental /src /target
```

Now, we're copying files from `/src` to `/target` as we were before, but we're now sending replaced files to the `/backup/incremental` directory. This means that when a file is going to be replaced on the target, the original file will be copied to `/backup/incremental`. This works because we used the `-b` option (backup) but we also used the `--backup-dir` option, which means that the replaced files won't be renamed, they'll simply be moved to the designated directory. This allows us to effectively perform incremental backups.

Building on our previous example, we can use the Bash shell itself to make incremental backups work even better. Consider these commands:

```
CURDATE=$(date +%m-%d-%Y)
export $CURDATE
sudo rsync -avb --delete --backup-dir=/backup/incremental/$CURDATE /src
/target
```

With this example, we grab the current date and store it in a variable (`CURDATE`). We'll also `export` the new variable so that it's fully available. In the `rsync` portion of the command, we use that variable for the `--backup-dir` option. This will copy the replaced files to a `backup` directory named after the `date` the command was run. Basically, if today's date was `08-17-2020`, the resulting command would be the same as if we had run the following:

```
sudo rsync -avb --delete --backup-dir=/backup/incremental/08-17-2020 /
src /target
```

Hopefully, you can see how flexible `rsync` is and how it can be used to not only copy files between directories and/or nodes, but also to serve as a backup solution as well (assuming you have a remote destination to copy files to). The best part is that this is only the beginning. If you consult the man page for `rsync`, you'll see that there are a lot of options you can use to customize it even further. Give it some practice, and you should get the hang of it in no time.

## Transferring files with SCP

A useful alternative to `rsync` is the **Secure Copy** (**SCP**) utility, which comes bundled with the OpenSSH client. It allows you to quickly copy files from one node to another. While `rsync` also allows you to copy files to other network nodes via SSH, SCP is more practical for one-off tasks; `rsync` is geared toward more complex jobs. If your goal is to send a single file or a small number of files to another machine, SCP is a great tool you can use to get the job done. If nothing else, it's yet another item for your administration toolbox. To utilize SCP, we'll use the `scp` command. Since you most likely already have the OpenSSH client installed, you should already have the `scp` command available. If you execute `which scp`, you should receive the following output:

```
/usr/bin/scp
```

If you don't see any output, make sure that the `openssh-client` package is installed.

Using SCP is very similar in nature to `rsync`. The command requires a source, a target, and a filename. To transfer a single file from your local machine to another, the resulting command would look similar to the following:

```
scp myfile.txt jdoe@192.168.1.50:/home/jdoe
```

With this example, we're copying the `myfile.txt` file (which is located in our current working directory) to a server located at `192.168.1.50`. If the target server is recognized by DNS, we could've used the DNS name instead of the IP address. The command will connect to the server as user `jdoe` and place the file into that user's `home` directory. Actually, we can shorten that command a bit:

```
scp myfile.txt jdoe@192.168.1.50:
```

Notice that I removed the target path, which was `/home/jdoe`. I'm able to omit the path to the target, since the `home` directory is assumed if you don't give the `scp` command a target path. Therefore, the `myfile.txt` file will end up in `/home/jdoe` whether or not I included the path to the `home` directory explicitly. If I wanted to copy the file somewhere else, I would definitely need to call out the location. Make sure you always include at least the colon when copying a file, since if you don't include it, you'll end up copying the file to your current working directory instead of the target.

The `scp` command also works in reverse:

```
scp jdoe@192.168.1.50:myfile.txt  .
```

With this example, we're assuming that `myfile.txt` is located in the `home` directory for the user `jdoe`. This command will copy that file to the current working directory of our local machine, since I designated the local path as a *single period* (which corresponds to our current working directory). Using `scp` in reverse isn't always practical, since you have to already know where the desired file is stored on the target before transferring it.

With our previous `scp` examples, we've only been copying a single file. If we want to transfer or download an entire directory and its contents, we will need to use the `-r` option, which allows us to do a recursive copy:

```
scp -r /home/jdoe/downloads/linux_iso jdoe@192.168.1.50:downloads
```

With this example, we're copying the local folder `/home/jdoe/downloads/linux_iso` to remote machine `192.168.1.50`. Since we used the `-r` option, `scp` will transfer the `linux_iso` folder and all of its contents. On the remote end, we're again connecting via the user `jdoe`. Notice that the target path is simply `downloads`. Since `scp` defaults to the user's `home` directory, this will copy the `linux_iso` directory from the source machine to the target machine under the `/home/jdoe/downloads` directory. The following command would've had the exact same result:

```
scp -r /home/jdoe/downloads/linux_iso jdoe@192.168.1.50:/home/jdoe/
downloads
```

The `home` directory is not the only assumption the `scp` command makes. It also assumes that SSH is listening on port `22` on the remote machine. Since it's possible to change the SSH port on a server to something else, port `22` may or may not be what's in use. If you need to designate a different port for `scp` to use, use the `-P` option:

```
scp -P 2222 -r /home/jdoe/downloads/linux_iso
jdoe@192.168.1.50:downloads
```

With that example, we're connecting to the remote machine via port `2222`. If you've configured SSH to listen on a different port, change the number accordingly.

Although port `22` is always the default for OpenSSH, it's common for some administrators to change it to something else. While changing the SSH port doesn't add a great deal of benefit in regard to security (an intensive port scan will still find your SSH daemon), it's a relatively easy change to make, and making it even just a little bit harder to find is beneficial. We'll discuss this further in *Chapter 21*, *Securing Your Server*.

Like most commands in the Linux world, the `scp` command supports verbose mode. If you want to see how the `scp` command progresses as it copies multiple files, add the `-v` option:

```
scp -rv /home/jdoe/downloads/linux_iso jdoe@192.168.1.50:downloads
```

Well, there you have it. The `scp` command isn't overly complex or advanced, but it's really great for situations in which you want to perform a one-time copy of a file from one node to another. Since it copies files over SSH, you benefit from its security, and it also integrates well with your existing SSH configuration. An example of this integration is the fact that `scp` recognizes your `~/.ssh/config` file (if you have one), so you can shorten the command even further. Go ahead and practice with it a bit, and in the next section, we'll go over yet another trick that OpenSSH has up its sleeve.

## Mounting remote directories with SSHFS

Earlier in this chapter, we took a look at several ways in which we can set up a Linux file server using Samba and/or NFS. There's another type of file-sharing solution I haven't mentioned yet, the **SSH Filesystem** (**SSHFS**). NFS and Samba are great solutions for designating file shares that are to be made available to other users, but these technologies may be more complex than necessary if you want to set up a temporary file-sharing service to use for a specific period of time. SSHFS allows you to mount a remote directory on your local machine, and have it treated just like any other directory. The mounted SSHFS directory will be available for the life of the SSH connection. When you're finished, you simply disconnect the SSHFS mount.

There are some downsides when it comes to SSHFS, however. First, the performance of file transfers won't be as fast as with an NFS mount, since there's encryption that needs to be taken into consideration as well. However, unless you're performing really resource-intensive work, you probably won't notice much of a difference anyway. Another downside is that you'd want to save your work regularly as you work on files within an SSHFS mount, because if the SSH connection drops for any reason, you may lose data. This logic is also true of NFS and Samba shares, but SSHFS is more of an on-demand solution and not something intended to remain connected and in place all the time.

To get started with SSHFS, we'll need to install it:

```
sudo apt install sshfs
```

Now we're ready to roll. For SSHFS to work, we'll need a directory on both your local Linux machine and a remote Linux server. SSHFS can be used to mount any directory from the remote server you would normally be able to access via SSH. That's really the only requirement. What follows is an example command to mount an external directory to a local one via SSHFS. In your tests, make sure to replace my sample directories with actual directories on your nodes, as well as using a valid user account:

```
sshfs myuser@192.168.1.50:/share/myfiles /mnt/myfiles
```

As you can see, the `sshfs` command is fairly straightforward. With this example, we're mounting `/share/myfiles` on `192.168.1.50` to `/mnt/myfiles` on our local machine. Assuming the command didn't provide an error (such as `access denied`, if you didn't have access to one of the directories on either side), your local directory should show the contents of the remote directory. Any changes you make to the files in the local directory will be made to the target. The SSHFS mount will basically function in the same way as if you had mounted an NFS or Samba share locally.

When we're finished with the mount, we should unmount it. There are two ways to do so. First, we can use the `umount` command as the `root` (just like we normally would):

```
sudo umount /mnt/myfiles
```

Using the `umount` command isn't always practical for SSHFS, though. The user that's setting up the SSHFS link may not have `root` permissions, which means that they won't be able to unmount it with the `umount` command. If you tried the `umount` command as a regular user, you would see an error similar to the following:

```
umount: /mnt/myfiles: Permission denied
```

It may seem rather strange that a normal user can mount an external directory via SSHFS, but not unmount it. Thankfully, there's a specific command a normal user can use, so we won't need to give them `root` or `sudo` access:

```
fusermount -u /mnt/myfiles
```

That should do it. With the `fusermount` command, we can unmount the SSHFS connection we set up, even without `root` access. The `fusermount` command is part of the **Filesystem in Userspace** (**FUSE**) suite, which is what SSHFS uses as its virtual filesystem to facilitate such a connection. The `-u` option, as you've probably guessed, is for unmounting the connection normally. There is also the `-z` option, which unmounts the SSHFS mount *lazily*. By lazily, I mean it basically unmounts the filesystem without any cleanup of open resources. This is a last resort that you should rarely need to use, as it could result in data loss.

Connecting to an external resource via SSHFS can be simplified by adding an entry for it in `/etc/fstab`. Here's an example entry using our previous example:

```
myuser@192.168.1.50:/share/myfiles    /mnt/myfiles    fuse.sshfs
rw,noauto,users,_netdev  0  0
```

Notice that I used the `noauto` option in the `fstab` entry, which means that your system will not automatically attempt to bring up this SSHFS mount when it boots. Typically, this is ideal. The nature of SSHFS is to create on-demand connections to external resources, and we wouldn't be able to input the password for the connection while the system is in the process of booting anyway. Even if we set up password-less authentication, the SSH daemon may not be ready by the time the system attempts to mount the directory, so it's best to leave the `noauto` option in place and just use SSHFS as the on-demand solution it is. With this `/etc/fstab` entry in place, any time we would like to mount that resource via SSHFS, we would only need to execute the following command going forward:

```
mount /mnt/myfiles
```

Since we now have an entry for `/mnt/myfiles` in `/etc/fstab`, the `mount` command knows that this is an SSHFS mount, where to locate it, and which user account to use for the connection. After you execute the example `mount` command, you should be asked for the user's SSH password (if you don't have password-less authentication configured) and then the resource should be mounted.

SSH sure does have a few unexpected tricks up its sleeve. Not only is it the de facto standard in the industry for connecting to Linux servers, but it also offers us a neat way of transferring files quickly and mounting external directories. I find SSHFS very useful in situations where I'm working on a large number of files on a remote server but want to work on them with applications I have installed on my local workstation. SSHFS allows us to do exactly that.

## Summary

In this chapter, we explored multiple ways of accessing remote resources. Just about every network has a central location for storing files, and we explored two ways of accomplishing this with NFS and Samba. Both NFS and Samba have their place in the data center and are very useful ways we can make resources on a server available to our users who need to access them. We also talked about `rsync` and `scp`, two great utilities for transferring data without needing to set up a permanent share. We closed off the chapter with a look at SSHFS, which is a very handy utility for mounting external resources locally, on demand.

Next up is *Chapter 13*, *Managing Databases*. Now that we have all kinds of useful services running on our Ubuntu Server network, it's only fitting that we take a look at serving databases as well. Specifically, we'll look at MariaDB. See you there!

## Further reading

- Ubuntu rsync documentation: `https://help.ubuntu.com/community/rsync`
- Ubuntu Samba documentation: `https://help.ubuntu.com/community/Samba`
- Ubuntu NFS documentation: `https://help.ubuntu.com/community/SettingUpNFSHowTo`
- Active Directory Integration: `https://ubuntu.com/server/docs/samba-active-directory`

# 13
# Managing Databases

The Linux platform has long been a very popular choice for hosting databases. Given the fact that databases power a large majority of popular websites across the internet nowadays, this is a very important role for servers to fill. Ubuntu Server is also a very popular choice for this purpose, as its stability is a major benefit to the hosting community. This time around, we'll take a look at MariaDB, a popular fork of MySQL. The goal won't be to provide a full walkthrough of MySQL's syntax (as that would be a full book in and of itself), but we'll focus on setting up and maintaining database servers utilizing MariaDB, and we'll even go over how to set up a primary/secondary relationship between them. If you already have a firm understanding of how to architect databases, you'll still benefit from this chapter as we'll be discussing Ubuntu's implementation of MariaDB in particular, which has its configuration organized a bit differently than in other platforms.

As we work through setting up our very own MariaDB server, we will cover the following topics:

- Preparations for setting up a database server
- Installing MariaDB
- Understanding the MariaDB configuration files
- Managing MariaDB databases
- Setting up a secondary database server

As with setting up any server for a particular purpose, we should first think about the goal and plan accordingly. So in the next section, we'll talk about some of the higher-level details to consider while preparing a database server.

## Preparations for setting up a database server

Before we get started with setting up our database server, there are a few odds and ends to get out of the way. As we go through this chapter, we'll set up a basic database server using MariaDB. I'm sure more than a few of my readers are familiar with MySQL. MySQL is a tried and true solution that is still in use in many data centers today, and that will probably continue to be the case for the foreseeable future. There's a good chance that a popular website or two that you regularly visit utilizes it on the backend. So, you may be wondering then, why not go over that instead of MariaDB?

There are two reasons why this book will focus on MariaDB. First, the majority of the Linux community is migrating over to it (more on that later), and it's also a drop-in replacement for MySQL. This means that any databases or scripts you've already written for MySQL will most likely work just fine with MariaDB, barring some edge cases. In reverse, the commands you practice with MariaDB should also function as you would expect on a MySQL server. This is great, considering that many MySQL installations are still in use in many data centers, and you'll be able to support those too. For the most part, there are very few reasons to stick with MySQL when your existing infrastructure can be ported over to MariaDB, and that's the direction the Linux community is headed toward anyway.

Why the change? If you regularly read the news regarding Linux topics, you may have seen articles from time to time regarding various distributions switching to MariaDB from MySQL. Red Hat is one such example; it switched to MariaDB in version 7 of Red Hat Enterprise Linux. Other distributions, such as Arch Linux and Fedora, went down the same route. This was partly due to a lack of trust in Oracle, the company that now owns MySQL. When Oracle became the owner of MySQL, there were some serious questions raised in the open source community regarding the future of MySQL as well as its licensing. I'm not going to get into any speculation about Oracle, the future of MySQL, or any politics regarding its future since it's not relevant to this book (and I'm not a fan of corporate drama). The fact, though, is that many distributions are moving toward MariaDB, and that seems to be the future. It's a great technology, and I definitely recommend it over MySQL for several reasons that are unrelated to current controversies.

MariaDB is more than just a fork of MySQL. On its own, it's a very competent database server. The fact that your existing MySQL implementations should be compatible with it eases adoption. But more than that, MariaDB makes some very worthwhile changes and improvements to MySQL that will only benefit you.

Everything you love about MySQL can be found in MariaDB, plus some cutting edge features that are exclusive to it. Some of the improvements in MariaDB include the fact that we'll receive faster security patches (since developers don't need to wait for approval from Oracle before releasing updates), as well as better performance. But, even better is the fact that MariaDB features additional clustering options that are leaps and bounds better and more efficient than plain old MySQL.

So, hopefully I've sold you on the value of MariaDB. Ultimately, whether or not you actually use it will depend on the needs of your organization. I've seen some organizations opt to stick with MySQL, if only for the sole reason that it's what they know. I can understand that if a solution has proven itself in your data center, there's really no reason to change if your database stack is working perfectly fine the way it is. To that end, while I'll be going over utilizing MariaDB, it's possible that the examples may work for MySQL as well. If in doubt, MariaDB is recommended for the examples in this chapter.

With regard to your server, a good implementation plan is key (as always). I won't spend too much time on this aspect, since by now I know you've probably been through a paragraph or two in this book where I've mentioned the importance of redundancy (and I'm sure I'll mention redundancy again a few more times before the last page). At this point, you're probably just setting up a lab environment or test server on which to practice these concepts before using your newfound skills in production. But when you do eventually roll out a database server into production, it's crucial to plan for long-term stability. Database servers should be regularly backed up, redundant (there I go again), and regularly patched. Later on in this chapter, I'll walk you through setting up a secondary database server, which will take care of the redundancy part. However, that's not enough on its own, as regular backups are important. There are many utilities that allow you to do this, such as `mysqldump`, and also take snapshots of your virtual machine (assuming you're not using a physical server). Both solutions are valid, depending on your environment. As someone who has lost an entire workday attempting to resurrect a fallen database server for a client (of which they had no backups or redundancy), my goal is simply to spare you that headache.

As far as how much resources a database server needs, that solely depends on your environment. MariaDB itself does not take up a huge amount of resources, but as with MySQL, your usage is dependent on your workload. Either you'll have a few dozen clients connecting, or a few thousand, or more. But one recommendation I'll definitely make is to use **Logical Volume Manager** (**LVM**) for the partition that houses your database files. This will certainly spare you grief in the long run. As we've discussed in *Chapter 9*, *Managing Storage Volumes*, LVM makes it very simple to expand a filesystem, especially on a virtual machine.

If your database server is on a virtual machine, you can add a disk to the volume group and expand it if your database partition starts to get full, and your customers will never notice there was ever about to be a problem. Without LVM, you'll need to shut down the server, add a new volume, `rsync` your database server files over to the new location, and then bring up the server. Depending on the size of your database, this situation can span hours. Do yourself a favor, use LVM.

With that out of the way, we can begin setting up MariaDB. For learning and testing purposes, you can use pretty much any server you'd like; physical, virtual, or **Virtual Private Server** (**VPS**). Once you're ready, let's move on and we'll get started!

## Installing MariaDB

Now we've come to the fun part, installing MariaDB. To get the ball rolling, we'll install the `mariadb-server` package:

```
sudo apt install mariadb-server
```

If your organization prefers to stick with MySQL, the package to install is `mysql-server` instead:

```
sudo apt install mysql-server
```

Although it might be tempting to try out both MySQL and MariaDB to compare and contrast their differences, I don't recommend switching from MariaDB to MySQL (or vice versa) on the same server. I've seen some very strange configuration issues occur on servers that had one installed and then were switched to the other (even after wiping the configuration). For the most part, it's best to pick one solution per server and stick with it. As a general rule, MySQL should be used if you have legacy databases to support. For brand-new installations, go with MariaDB.

Going forward, I'll assume that you've installed MariaDB, though the instructions here shouldn't differ much between them. For the name of the service, you'll want to substitute `mariadb` for `mysql` whenever I mention it if you are using MySQL instead. However, keep in mind that compatibility with MySQL cannot be guaranteed. Previous versions of Ubuntu Server used `mysql` for the service name, even when installing MariaDB (Ubuntu 16.04 is an example of this). This is just something to keep in mind if you're supporting a legacy edition of Ubuntu.

After you install the `mariadb-server` package, check to make sure the service started and is enabled. By default, it should already be running:

```
systemctl status mariadb
```

Next, we'll want to add some security to our MariaDB installation (even though we're using MariaDB, the name of the following command hasn't been updated and still includes `mysql` in the name):

```
sudo mysql_secure_installation
```

At this point, we haven't set a `root` password yet, so go ahead and just press *Enter* when the script asks for it. This script will ask you additional questions. Next, it will ask you if you want to set a `root` password. The `root` user for MariaDB is not the same as the `root` user on your system, and you definitely should create a password for it. So when this comes up, press *y* to tell it you want to create a `root` password, then enter that password twice.

After setting the `root` password, the script will ask you whether you'd like to remove anonymous users, and also disallow remote access to the database server. You should answer yes to both. The latter is especially important, as there's almost never a situation in which allowing external access to MySQL/MariaDB is a good idea. Even if you're hosting a website for external users, those users only need access to the website, not the database server. The website itself will interface with the database locally as needed; an external connection wouldn't be necessary. Basically, just answer `yes` to everything the script asks you.

The entire process after executing `mysql_secure_installation` looks like the following:

```
Enter current password for root (enter for none):
Set root password? [Y/n]
Remove anonymous users? [Y/n]
Disallow root login remotely? [Y/n]
Remove test database and access to it? [Y/n]
Reload privilege tables now? [Y/n]
```

At this point, we officially have a fully functional database server. The previous command allowed us to apply some basic security, and our database server is now available to us. To connect to it and manage it, we'll use the `mariadb` command to access the MariaDB shell, where we'll enter commands to manage our database(s). There are actually two methods to connect to this shell. The first method is by simply using the `mariadb` command with `sudo`:

```
sudo mariadb
```

This particular command works because if you use the `mariadb` command as `root` (we used `sudo` in this example) the password is bypassed. In fact, we didn't even enter the username either; `root` is assumed if you are attempting to access MariaDB with `sudo`. This is by far the simplest way to connect. However, some of you may be accustomed to a different method of authentication if you've used other Linux distributions: entering the username and password. In that case, the command will look like this (it won't work by default though):

```
mariadb -u root -p
```

When that command works correctly, it will ask you for your `root` password and then let you into the shell. However, by default, the `root` user is set up to use a completely different mode of authentication altogether (Unix sockets) and this will fail with the following error (even if you enter the correct password):

```
ERROR 1698 (28000): Access denied for user 'root'@'localhost'
```

You have two options for dealing with this. First, you can simply use `sudo mariadb` to access the MariaDB shell and not use the other method. Doing so is perfectly valid, and there are no downsides as it gives you the same level of access. If you prefer to access the `root` account via the traditional means (by providing the username and password for the `root` user) you'll need to change the `root` user to use the native password authentication method instead. To do so, first access the MariaDB shell and then enter these two commands:

```
UPDATE mysql.user SET plugin = 'mysql_native_password' WHERE
USER='root';
FLUSH PRIVILEGES;
```

Now, you'll be able to access the MariaDB shell as root with native authentication:

```
mariadb -u root -p
```

It's recommended to create a different user in order to manage your MariaDB installation, as logging in to root is not recommended in most cases. We'll be creating additional users later on in this chapter, but for now, the root account is the only one we have available at the moment. It's common practice to use the root account to do the initial setup, and then create a different user for administrative purposes going forward. However, the root account is still often used for server maintenance, so use your best judgment.

So, now that we have access to the MariaDB shell, what can we do with it? The commands we'll execute on this shell allow us to do things such as create and delete databases and users, add tables, and so on. The `mariadb` command comes from the `mariadb-client-10.3` package, which was installed as a dependency when we installed `mariadb-server`. Entering the `mariadb` command by itself with no options connects us to the database server on our local machine. This utility also lets us connect to external database servers to manage them remotely, which we'll discuss later.

The MariaDB shell prompt will look like this:

```
MariaDB [(none)]>
```

We'll get into MariaDB commands and user management later. For now, you can exit the shell. To exit, you can type `exit` and press *Enter* or press *Ctrl* + *d* on your keyboard.

Now, our MariaDB server is ready to go. While you can now move on to the next section, you might want to consider setting up another MariaDB server by following these steps on another machine. If you have room for another virtual machine, it might be a good idea for you to get this out of the way now, since we'll be setting up a secondary database server later.

## Understanding the MariaDB configuration files

Now that we have MariaDB installed, let's take a quick look at how its configuration is stored. While we won't be changing much of the configuration in this chapter (aside from adding parameters related to setting up a secondary database instance), it's a good idea to know where to find the configuration, since you'll likely be asked by a developer to tune the database configuration at some point in your career. This may involve changing the storage engine, buffer sizes, or countless other settings. A full walkthrough on performance tuning is outside the scope of this book, but it will be helpful to know how the settings for MariaDB are read, since Ubuntu's implementation is fairly unique.

The configuration files for MariaDB are stored in the `/etc/mysql` directory. In that directory, you'll see the following files by default:

```
debian.cnf
debian-start
mariadb.cnf
my.cnf
my.cnf.fallback
```

You'll also see the following directories:

```
conf.d
mariadb.conf.d
```

The configuration file that MariaDB reads on startup is the `/etc/mysql/mariadb.cnf` file. This is where you'll begin perusing when you want to configure the daemon, but we'll get to that soon. The `/etc/mysql/debian-start` file is actually a script that sets default values for MariaDB when it starts, such as setting some environment variables. It also defines a task that is executed if the `mariadb` process dies or exits, and allows it to check for crashed tables.

The `debian-start` script also loads the `/etc/mysql/debian.cnf` file, which sets some client settings for the `mariadb` daemon. Here's a list of values from that file:

```
[client]
host     = localhost
user     = root
password =
socket   = /var/run/mysqld/mysqld.sock
[mysql_upgrade]
host     = localhost
user     = root
password =
socket   = /var/run/mysqld/mysqld.sock
basedir  = /usr
```

The defaults for these values are fine and there's rarely a reason to change them. Essentially, the file sets the default user, host, and socket location. If you've used MySQL before on other platforms, you may have seen many of those settings in the `/etc/my.cnf` file, which is typically the standard file for the `mariadb` daemon. With MariaDB on Ubuntu Server, you can see that the default layout of the files was changed considerably.

The `/etc/mysql/mariadb.cnf` file sets the global defaults for MariaDB. However, in Ubuntu's implementation, this default file just includes configuration files from the `/etc/mysql/conf.d` and `/etc/mysql/mariadb.conf.d` directories. Within those directories, there are additional files ending with the `.cnf` extension. Many of these files contain default configuration values that would normally be found in a single file, but Ubuntu's implementation modularizes these settings into separate files instead. For our purposes in this book, we'll be editing the `/etc/mysql/conf.d/mysql.cnf` file when it is time to set up a relationship between primary and secondary servers.

The other configuration files aren't relevant for the content of this book, and their current values are more than sufficient for what we need. When it comes to performance tuning, you may consider creating a new configuration file ending with the `.cnf` extension, with specific tuning values as provided by the documentation of a software package you want to run that interfaces with a database, or requirements given to you by a developer.

For additional information on how these configuration files are read, you can refer to the `/etc/mysql/mariadb.cnf` file, which includes some helpful content at the top of the file that details the order in which these configuration files are read, as well as their purpose. Here's an excerpt of these comments from that file:

```
# The MariaDB/MySQL tools read configuration files in the following
order:
# 1. "/etc/mysql/mariadb.cnf" (this file) to set global defaults,
# 2. "/etc/mysql/conf.d/*.cnf" to set global options.
# 3. "/etc/mysql/mariadb.conf.d/*.cnf" to set MariaDB-only options.
# 4. "~/.my.cnf" to set user-specific options.
```

As we can see, when MariaDB starts up, it first reads the `/etc/mysql/mariadb.cnf` file, followed `by .cnf` files stored within the `/etc/mysql/conf.d` directory, then the `.cnf` files stored within the `/etc/mysql/mariadb.conf.d` directory, followed by any user-specific settings stored within a `.my.cnf` file that may be present in the user's `home` directory.

With Ubuntu's implementation, when the `/etc/mysql/mariadb.cnf` file is read during startup, the process will immediately scan the contents of `/etc/mysql/conf.d` and `/etc/mysql/mariadb.conf.d`, because the `/etc/mysql/mariadb.cnf` file contains the following lines:

```
!includedir /etc/mysql/conf.d/
!includedir /etc/mysql/mariadb.conf.d/
```

As you can see, the order that the configuration files are checked is set to the `mariadb.cnf` file first, followed by the `/etc/mysql/conf.d` and `/etc/mysql/mariadb.conf.d` directories.

This may be a bit confusing at first, because the default configuration for MariaDB in Ubuntu Server essentially consists of files that redirect to other files. But the main takeaway is that any configuration changes you make that are not exclusive to MariaDB (basically, configuration compatible with MySQL itself) should be placed in a configuration file that ends with the `.cnf` extension, and then stored in the `/etc/mysql/conf.d` directory.

If the configuration you're wanting to add is for a feature exclusive to MariaDB (but not compatible with MySQL itself), the configuration file should be placed in the `/etc/mysql/mariadb.conf.d` directory instead. For our purposes, we'll be editing the `/etc/mysql/conf.d/mysql.cnf` file when it comes time to set up our primary/ secondary replication, since the method we'll be using is not specific to MariaDB.

In this section, we discussed MariaDB configuration and how it differs from its implementation in other platforms. The way the configuration files are presented is not the only difference in Ubuntu's implementation of MariaDB; there are other differences as well. In the next section, we'll take a look at a few additional ways in which Ubuntu's implementation differs from implementations in other distributions of Linux.

## Managing MariaDB databases

Now that our MariaDB server is up and running, we can finally look into managing it. In this section, I'll demonstrate how to connect to a database server using the `mariadb` command, which will allow us to create databases, remove (drop) them, and also manage users and permissions.

To begin, we'll need to create an administrative user for MariaDB. The `root` account already exists as the default administrative user, but it's not a good idea to allow others to use that account. Instead, it makes more sense to create an administrative account separate from `root` for managing our databases. Therefore, we'll begin our discussion on managing databases with user management. The users we'll manage within MariaDB are specific to MariaDB; these are separate from the user accounts on the actual system.

To manage and interact with databases, we'll need to enter the MariaDB shell, and the same goes when it comes to creating database users. Right now, since we only have the `root` account, we'll need to access the current MariaDB implementation as `root` in order to set up our administrative user. If you've set up standard authentication, as we discussed earlier in this chapter, you can access the prompt using the standard means:

```
mariadb -u root -p
```

If you haven't switched the `root` account to standard authentication, you can simply access `mariadb` with `sudo`:

```
sudo mariadb
```

Alternatively, we can use the following command as a normal user to switch to `root`, without using `sudo`:

```
mariadb
```

Once inside the MariaDB shell, your prompt will change to the following:

```
MariaDB [(none)]>
```

Now, we can create our new administrative user. I'll call mine `admin` in my examples, but you can use whatever name you'd like. In a company I used to work for, we used the username `velociraptor` as our administrative user on our servers, since nothing is more powerful than a velociraptor (and they can open doors). Feel free to use a clever name, but just make sure you remember it. Using a non-standard username has the added benefit of security by obscurity; the name wouldn't be what an intruder would expect.

Here's the command to create a new user in MariaDB (replace the username and password in the command with your desired credentials):

```
CREATE USER 'admin'@'localhost' IDENTIFIED BY 'password';
FLUSH PRIVILEGES;
```

When it comes to MySQL syntax, the commands are not case sensitive (though the data parameters are), but it's common to capitalize instructions to separate them from data. During the remainder of this chapter, we'll be executing some commands within the Linux shell, and others within the MariaDB shell. I'll let you know which shell each command needs to be executed in as we come to them, but if you are confused, just keep in mind that MariaDB commands are the only ones that are capitalized.

With the preceding commands, we're creating the `admin` user and restricting it to `localhost`. This is important because we don't want to open up the `admin` account to the world. We're also flushing privileges, which causes MariaDB to reload its privileges information. The `FLUSH PRIVILEGES` command should be run every time you add a user or modify permissions. I may not always mention the need to run this command, so you might want to make a mental note of it and make it a habit now.

As I mentioned, the previous command created the admin user but is only allowing it to connect from localhost. This means that an administrator would first need to log in to the server itself before they would be able to log in to MariaDB with the admin account. As an example of the same command (but allowing remote login from any other location), the following command is a variation that will do just that:

```
CREATE USER 'admin'@'%' IDENTIFIED BY 'password';
```

Can you see the percent symbol (%) in place of localhost? That basically means *everywhere*, which indicates we're creating a user that can be logged in to from any source (even external nodes). However, by restricting our user to localhost with the first command, we're making our server just a bit more secure. You can also restrict access to particular networks, which is desired if you really do need to allow a database administrator access to the server remotely:

```
CREATE USER 'admin'@'192.168.1.%' IDENTIFIED BY 'password';
```

That's a little better, but not as secure as limiting login to localhost. As you can see, the % character is basically a wildcard, so you can restrict access to needing to be from a specific IP or even a particular subnet.

So far, all we did is create a new user; we have yet to give this user any permissions. We can create a set of permissions (also known as **grants**) with the GRANT command. First, let's give our admin user full access to the database server when called from localhost:

```
GRANT ALL PRIVILEGES ON *.* TO 'admin'@'localhost';
FLUSH PRIVILEGES;
```

Now, we have an administrative user we can use to manage our server's databases. We can use this account for managing our server instead of the root account. Any logged-on Linux user will be able to access the database server and manage it, provided they know the password. To access the MariaDB shell as the admin user that we created, the following command will do the trick:

```
mariadb -u admin -p
```

After entering the password, you'll be logged in to MariaDB as admin.

In addition, you can actually provide the password to the mariadb command without needing to be prompted for it:

```
mariadb -u admin -p<password>
```

Notice that there is no space in between the `-p` option and the actual password (though it's common to put a space between the `-u` option and the username). As useful as it is to provide the username and password in one shot, I don't recommend that you ever use that method. This is because any Linux command you type is saved in the history, so anyone can view your command history and they'll see the password in plain text. I only mention it here because I find that many administrators do this, even though they shouldn't. At least now you're aware of this method and why it's wrong.

The `admin` account we created is only intended for system administrators who need to manage databases on the server. The password for this account should not be given to anyone other than staff employees or administrators that absolutely need it. Additional users can be added to the MariaDB server, each with differing levels of access. Keep in mind that our `admin` account can manage databases but not users. This is important, as you probably shouldn't allow anyone other than server administrators to create users. You'll still need to log in as `root` to manage user permissions.

It may also be useful to create a read-only user for MariaDB for employees who need to be able to read data but not make changes. Back in the MariaDB shell (as `root`), we can issue the following command to effectively create a read-only user:

```
GRANT SELECT ON *.* TO 'readonlyuser'@'localhost' IDENTIFIED BY
'password';
```

With this command (and flushing privileges afterward), we've done two things. First, we created a new user and also set up grants for that user with a single command. Second, we created a read-only user that can view databases but not manage them (we restricted the permissions to `SELECT`). This is more secure. In practice, it's better to restrict a read-only user to a specific database. This is typical in a development environment, where you'll have an application that connects to a database over the network and needs to read information from it. We'll go over this scenario soon.

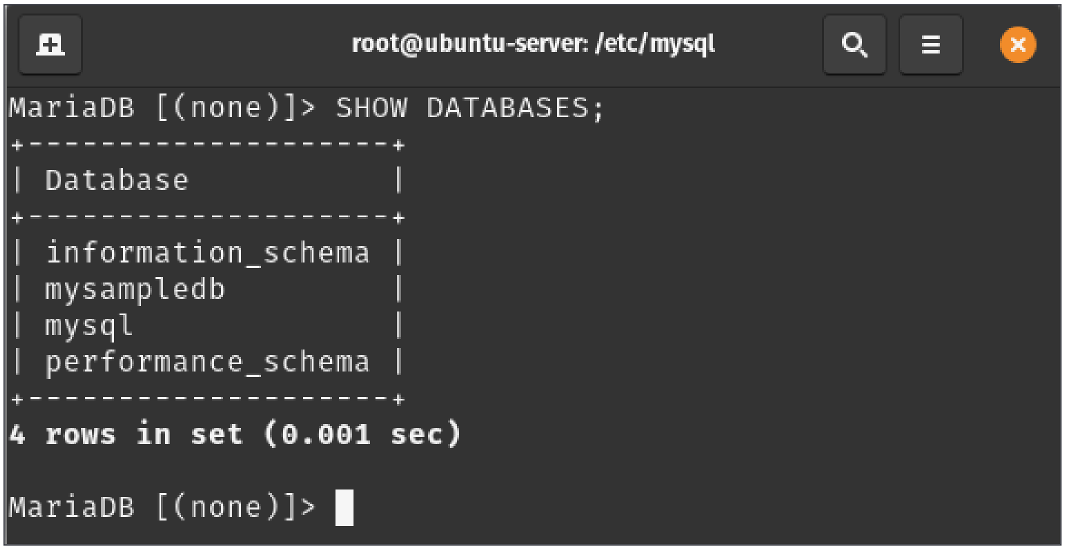
Next, let's create a database. At the MariaDB prompt, execute:

```
CREATE DATABASE mysampledb;
```

That was easy. We should now have a database on our server named `mysampledb`. To list all databases on our server (and confirm our database was created properly), we can execute the following command:

```
SHOW DATABASES;
```

This will produce an output like the following:



root@ubuntu-server: /etc/mysql

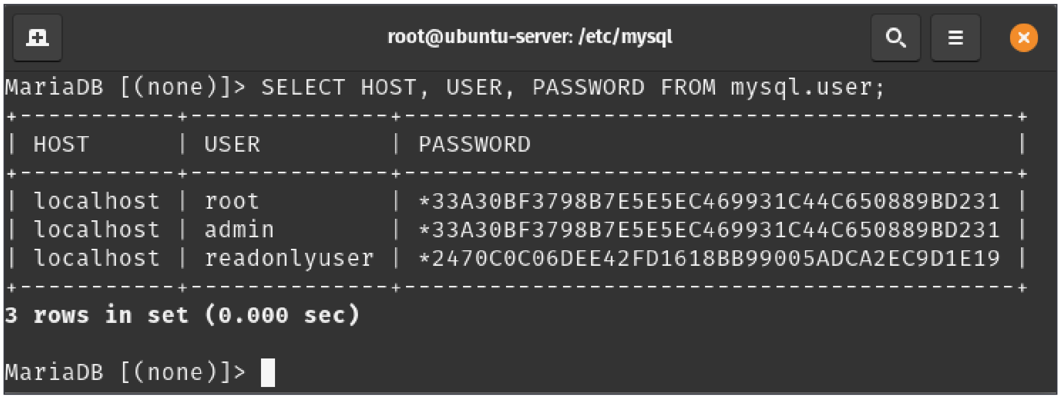
```
MariaDB [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysampledb         |
| mysql              |
| performance_schema |
+--------------------+
4 rows in set (0.001 sec)

MariaDB [(none)]>
```

Figure 13.1: Listing MariaDB databases

The output will show some system databases that were created for us, but our new database should be listed among them. We can also list users just as easily:

```
SELECT HOST, USER, PASSWORD FROM mysql.user;
```

Entering this command will result in something similar to the following output:



root@ubuntu-server: /etc/mysql

```
MariaDB [(none)]> SELECT HOST, USER, PASSWORD FROM mysql.user;
+-----------+--------------+-------------------------------------------+
| HOST      | USER         | PASSWORD                                  |
+-----------+--------------+-------------------------------------------+
| localhost | root         | *33A30BF3798B7E5E5EC469931C44C650889BD231 |
| localhost | admin        | *33A30BF3798B7E5E5EC469931C44C650889BD231 |
| localhost | readonlyuser | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
+-----------+--------------+-------------------------------------------+
3 rows in set (0.000 sec)

MariaDB [(none)]>
```

Figure 13.2: Listing MariaDB users

In a typical scenario, when installing an application that needs its own database, we'll create the database and then a user for that database. We'll normally want to give that user permission to only that database, with only as much permission as required to allow it to function properly. We've already created the `mysampledb` database, so if we want to create a user with read-only access to it, we can do so with the following command:

```
GRANT SELECT ON mysampledb.* TO 'appuser'@'localhost' IDENTIFIED BY
'password';
```

With one command, we're not only creating the user `appuser`, but we're also setting a password for it, in addition to allowing it to have `SELECT` permissions on the `mysampledb` database. This is equivalent to read-only access. If our user needed full access, we could use the following instead:

```
GRANT ALL ON mysampledb.* TO 'appuser'@'localhost' IDENTIFIED BY
'password';
```

To double-check that we've executed the command correctly, we can use this command to show the grants for a particular user:

```
SHOW GRANTS FOR 'appuser'@'localhost';
```

Now, our `appuser` has full access but only to the `mysampledb` database. Of course, we should only provide full access to the database if absolutely necessary. We can also provide additional permissions, such as `DELETE` (whether or not the user has permission to delete rows from database tables), `CREATE` (which controls whether the user can add rows to the database), `INSERT` (controls whether or not the user can add new rows to a table), `SELECT` (allows the user to read information from the database), `DROP` (allows the user to fully remove a database), and `ALL` (which gives the user everything). There are other permissions we can grant or deny; check the MariaDB documentation for more details. The types of permissions you'll need to grant to a user to satisfy the application you're installing will depend on the documentation for that software. Always refer to the installation instructions for the application you're attempting to install to determine which permissions are required for it to run.

If you'd like to remove user access, you can use the following command to do so (substituting `myuser` with the user account you wish to remove and `host` with the proper host access you previously granted the user):

```
DELETE FROM mysql.user WHERE user='myuser' AND host='localhost';
```

Now, let's go back to databases. Now that we've created the `mysampledb` database, what can we do with it? We'll add tables and rows, of course! A database is useless without actual data, so we can work through some examples of adding data to our database to see how this works. First, log in to the MariaDB shell as a user with full privileges to the `mysampledb` database. Now, we can have some fun and modify the contents. Here are some examples you can follow:

```
USE mysampledb;
```

The `USE` command allows us to select a database we want to work with. The MariaDB prompt will change from `MariaDB [(none)]>` to `MariaDB [mysampledb]>`. This is very useful, as the MariaDB prompt changes to indicate which database we are currently working with. We basically just told `MariaDB` that for all of the commands we're about to execute, we would like them used against the `mysampledb` database.

Now, we can `CREATE` a table in our database. It doesn't matter what you call yours, since we're just practicing. I'll call mine `Employees`:

```
CREATE TABLE Employees (Name char(15), Age int(3), Occupation
char(15));
```

We can verify this command by showing the columns in the database, to ensure it shows what we expect:

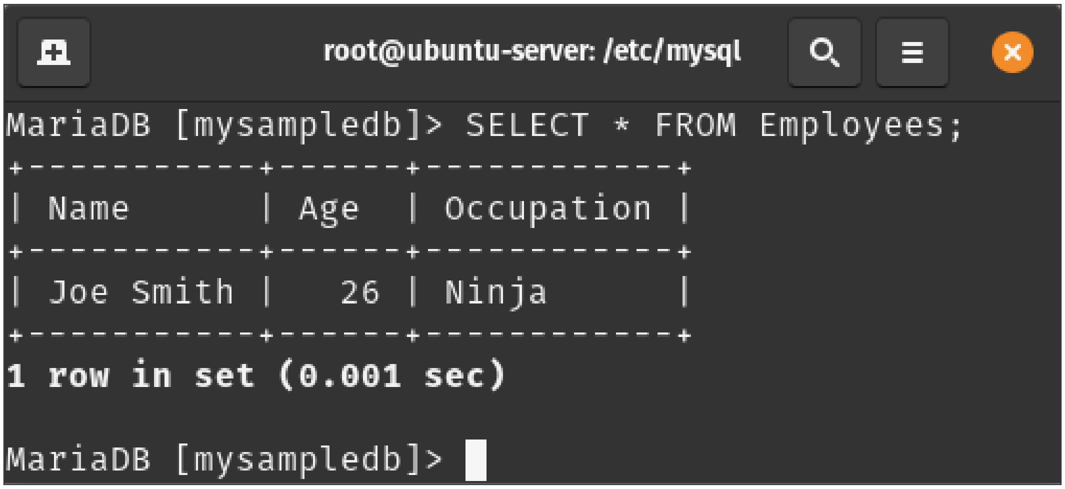
```
SHOW COLUMNS IN Employees;
```

With this command, we've created a table named `Employees` that has three columns (`Name`, `Age`, and `Occupation`). To add new data to this table, we can use the following `INSERT` command:

```
INSERT INTO Employees VALUES ('Joe Smith', '26', 'Ninja');
```

The example `INSERT` command adds a new employee to our `Employees` table. When we use `INSERT`, we insert all the data for each of the columns. Here, we have an employee named `Joe`, who is `26` years old and whose occupation is a `Ninja`. Feel free to add additional employees; all you would need to do is formulate additional `INSERT` statements and provide data for each of the three fields. When you're done, you can use the following command to show all of the data in this table:

```
SELECT * FROM Employees;
```



```
MariaDB [mysampledb]> SELECT * FROM Employees;
+-----------+------+------------+
| Name      | Age  | Occupation |
+-----------+------+------------+
| Joe Smith |   26 | Ninja      |
+-----------+------+------------+
1 row in set (0.001 sec)

MariaDB [mysampledb]>
```

Figure 13.3: Listing database rows from a table

To remove an entry, the following command will do what we need:

```
DELETE FROM Employees WHERE Name = 'Joe Smith';
```

Basically, we're using the `DELETE FROM` command, giving the name of the table we wish to delete from (`Employees`, in this case), and then using `WHERE` to provide some search criteria for narrowing down our command.

The `DROP` command allows us to delete tables or entire databases, and it should be used with care. I don't actually recommend you delete the database we just created, since we'll use it for additional examples. But if you really wanted to drop the `Employees` table, you could use:

```
DROP TABLE Employees;
```

Or use this to drop the entire database:

```
DROP DATABASE mysampledb;
```

There is, of course, much more to MariaDB and its MySQL syntax than the samples I have provided, but this should be enough to get you through the examples in this book. As much as I would love to give you a full walkthrough of the MySQL syntax, it would easily push this chapter beyond a reasonable number of pages. If you'd like to push your skills beyond the samples of this chapter, there are great books available that are dedicated to the subject.

Before I close this section though, I think it will be worthwhile for you to see how to back up and restore your databases. To do this, we have the mysqldump command at our disposal. Its syntax is very simple, as you'll see. First, exit the MariaDB shell and return to your standard Linux shell. Since we've already created an admin user earlier in the chapter, we'll use that user for the purposes of our backup:

```
mysqldump -u admin -p --databases mysampledb > mysampledb.sql
```

With this example, we're using mysqldump to create a copy of the mysampledb database and storing it in a file named mysampledb.sql. Since MariaDB requires us to log in, we authenticate to MariaDB using the -u option with the username admin and the -p option, which will prompt us for a password. The --databases option is necessary because, by default, mysqldump does not include the database create statement nor the tables. However, the --databases option forces this, which just makes it easier for you to restore. Assuming that we were able to authenticate properly, the contents of the mysampledb database will be dumped into the mysampledb.sql file. This export should happen very quickly, since this database probably only contains a single table and a few rows. Larger production databases can take hours to dump.

Restoring a backup is fairly simple. We can utilize the mariadb command with the backup file used as a source of input:

```
sudo mariadb -u admin -p < mysampledb.sql
```

So, there you have it. The mysqldump command is definitely very handy in backing up databases. In the next section, we'll work through setting up a secondary database server.

## Setting up a secondary database server

Redundancy is an amazing thing. If a primary server fails for some reason, you can keep your applications running by having a secondary database server available in case the original meets its demise. Of course, you can always create regular backups of your database servers and restore if necessary, but it's very hard to keep up with databases that are always changing, so backups have a tendency to become stale quite fast. A secondary database server enables you to have a copy that is always up to date. This doesn't mean that you no longer need backups, but it does give you another option for recovery when faced with a problem.

The industry is moving away from terms like "Master" and "Slave" to describe a primary and secondary server. In this chapter and moving forwards, we will use the terms "Primary" and "Secondary" to describe the relationship of one main database server, that replicates to another. Since this change started after Ubuntu 20.04 was released, the commands and configuration inside Ubuntu 20.04 still use `master` and `slave` when the relationships of the servers are described. Therefore, the verbiage around our actions will use "primary" and "secondary," even though the verbiage inside Ubuntu 20.04 still uses the old terminology. Just keep in mind that the naming will likely switch to a new naming scheme in the future.

At this point, we have one database server already. To set up a secondary database instance, all you really need in order to begin the process is to set up another physical server or virtual machine and install the `mariadb-server` package as we did earlier. If you've already set up two database servers as recommended earlier in the chapter, you're ready to begin. If not, feel free to spin up another virtual machine and follow the process from the *Installing MariaDB* section, which covered the initial setup of MariaDB. Go ahead and set up another server if you haven't already done so. Of your two servers, one should be designated as the primary and the other as the secondary, so make a note of the IP addresses for each.

To begin, we'll first start working on the primary. We'll need to edit the `/etc/mysql/conf.d/mysql.cnf` file on the server you wish to be the primary. Currently, the file contains just the following line:

```
[mysql]
```

Right underneath that, add a blank line and then the following code:

```
[mysqld]
log-bin
binlog-do-db=mysampledb
server-id=1
```

With this configuration, we're first enabling binary logging, which is required for a primary/secondary server to function properly. Binary logs contain records of all database changes, which enables a secondary database instance to reproduce changes made to the primary server. These binary logs record changes made to a database, which will then be transferred to a secondary server.

Another configuration file that we'll need to edit is `/etc/mysql/mariadb.conf.d/50-server.cnf`. In this file, we have the following line:

```
bind-address = 127.0.0.1
```

With this default setting, the `mysql` daemon is only listening for connections on localhost (`127.0.0.1`), which is a problem since we'll need to connect to it from another machine (the secondary server). Change this line to the following:

```
bind-address = 0.0.0.0
```

Next, we'll need to access the MariaDB shell on the primary server and execute the following commands:

```
GRANT REPLICATION SLAVE ON *.* to 'replicate'@'192.168.1.204'
identified by 'password';
```

Here, we're creating a replication user named `replicate` and allowing it to connect to our primary server from the IP address `192.168.1.204`. Be sure to change that IP to match the IP of your secondary server, but you can also use a hostname identifier such as `%.mydomain` if you have a domain configured, which is equivalent to allowing any hostname that ends with `.mydomain`. Also, we're setting the password for this user to `password`, so feel free to customize that as well to fit your password requirements (be sure to make a note of the password).

We should now restart the `mariadb` daemon so that the changes we've made to the `mysql.cnf` file take effect:

```
sudo systemctl restart mariadb
```

Next, we'll set up the secondary server. But before we do that, there's a consideration to make now that will possibly make the process easier on us. In a production environment, it's very possible that data is still being written to the primary server. The process of setting up a secondary server is much easier if we don't have to worry about the primary database changing while we set up the secondary. The following command, when executed within the MariaDB shell, will lock the database and prevent additional changes:

```
FLUSH TABLES WITH READ LOCK;
```

If you're absolutely sure that no data is going to be written to the primary server, you can disregard that step.

Next, we should utilize `mysqldump` to make sure both the primary and the secondary servers contain the same data before we start synchronizing them. The process is smoother if we begin with them already synchronized, rather than trying to mirror the databases later. Using `mysqldump` as we did in the previous section, create a dump of the primary server's database and then import that dump into the secondary. The easiest way to transfer the dump file is to use `rsync` or `scp`. Then, on the secondary instance, use `mariadb` to import the file.

The command to back up the database on the primary server becomes the following:

```
mysqldump -u admin -p --databases mysampledb > mysampledb.sql
```

After transferring the `mysampledb.sql` file to the secondary server, you can import the backup into the secondary server:

```
mariadb -u root -p < mysampledb.sql
```

Also on the secondary server, we'll need to edit `/etc/mysql/conf.d/mysql.cnf` and then place the following code at the end (make sure to add a blank line after `[mysql]`:

```
[mysqld]
server-id=2
```

Although it's outside the scope of this book, you can set up more than just one secondary database server. If you do, each will need a unique `server-id`.

Make sure you restart the `mariadb` unit on the secondary server before continuing:

```
sudo systemctl restart mariadb
```

From the `root` MariaDB shell on your secondary server, enter the following command. Change the IP address in the command accordingly:

```
CHANGE MASTER TO MASTER_HOST="192.168.1.184", MASTER_USER='replicate',
MASTER_PASSWORD='password';
```

Now that we're finished configuring the synchronization, we can unlock the primary server's tables. On the primary server, execute the following command within the MariaDB shell:

```
UNLOCK TABLES;
```

Now, we can check the status of the secondary server to see whether or not it is running.

Within the secondary server's MariaDB shell, execute the following command:

```
SHOW SLAVE STATUS \G;
```

Here, we're adding `\G`, which changes the output to be displayed vertically instead of horizontally.

Assuming all went well, we should see the following line in the output:

```
Slave_IO_State: Waiting for master to send event
```

If the secondary server isn't running (`Slave_IO_State` is blank), execute the following command:

```
START SLAVE;
```

Next, check the status of the secondary server process again to verify:

```
SHOW SLAVE STATUS \G;
```

From this point forward, any data you add to your database on the primary server should be replicated to the secondary. To test, add a new record to the `Employees` table on the `mysampledb` database on the primary server:

```
USE mysampledb;
INSERT INTO Employees VALUES ('Optimus Prime', '100', 'Transformer');
```

On the secondary server, check the same database and table for the new value to appear. It may take a second or two:

```
USE mysampledb;
SELECT * FROM Employees;
```

If you see any errors in the `Slave_IO_State` line when you run `SHOW SLAVE STATUS \G`, or your databases aren't synchronizing properly, here are a few things you can try. First, make sure that the primary database server is listening for connections on `0.0.0.0` port `3306`. To test this, run this variation of the `ss` command to see which port the `mariadb` process is listening on (it's listed as `mysqld`):

```
ss -tulpn | grep mysqld
```

The output should be similar to the following:

```
tcp   LISTEN 0      80                              0.0.0.0:3306
0.0.0.0:*            users:(("mysqld",pid=9768,fd=23))
```

If you see that the service is listening on `127.0.0.1:3306` instead, that means it's only accepting connections from localhost. Earlier in this section, I mentioned changing the `bind` address in the `/etc/mysql/mariadb.conf.d/50-server.cnf` file. Make sure you've already done that and restart `mariadb`. During my tests, I've actually had one situation where the `mariadb` service became locked after I made this change, and attempting to restart the process did nothing (I ended up having to reboot the entire server, which is not typically something you'd have to do). Once the server came back up, it was listening for connections from the network.

If you receive errors on the secondary server when you run `SHOW SLAVE STATUS \G`, with regards to authentication, make sure you've run `FLUSH PRIVILEGES` on the primary server. Even if you have, run it again to be sure. Also, double-check that you're synchronizing with the correct username, IP address, and password. For your convenience, here's the command we ran on the primary server to grant replication permissions:

```
GRANT REPLICATION SLAVE ON *.* to 'replicate'@'192.168.1.204'
identified by 'password';
FLUSH PRIVILEGES
```

Here's the command that we ran on the secondary server:

```
CHANGE MASTER TO MASTER_HOST="192.168.1.184", MASTER_USER='replicate',
MASTER_PASSWORD='password';
```

Finally, make sure that your primary database and the secondary database both contain the same databases and tables. The primary server won't be able to update a database on the secondary server if it doesn't exist there. Flip back to my example usage on `mysqldump` if you need a refresher. You should only need to use `mysqldump` and import the database onto the secondary server once, since after you get the replication going, any changes made to the database on the primary server should follow over to the secondary. If you have any difficulties with the `mysqldump` command, you can manually create `mysampledb` and the `Employees` table on the secondary server, which is really all it needs for synchronization to start.

Synchronization should then begin within a minute, but you can execute `STOP SLAVE`, followed by `START SLAVE`, on the secondary server to force it to try to synchronize again without waiting.

And that should be all there is to it. At this point, you should have fully functional primary and secondary database servers at your disposal. To get additional practice, try adding additional databases, tables, and users, and insert new rows into your databases. It's worth mentioning that the users we've created here will not be synced to the secondary server, so you can use the commands we used earlier in this chapter to create users on the secondary server if you wish for them to be present there.

## Summary

Depending on your skillset, you're either an administrator who is learning about SQL databases for the first time, or you're a seasoned veteran who is curious about how to implement a database server with Ubuntu Server. In this chapter, we dove into Ubuntu's implementation of this technology and worked through setting up our own database server. We also worked through some examples of the MariaDB syntax, such as creating databases, as well as setting up users and their grants. We also worked through setting up a primary and secondary server for replication.

Database administration is a vast topic, and we've only scratched the surface here. Being able to manage MySQL and MariaDB databases is a very sought-after skill for sure. If you haven't worked with these databases before, this chapter will serve as a good foundation for you to start your research.

In the next chapter, we'll use our database server to act as a foundation for Nextcloud, which we will set up as part of our look into setting up a web server. When you've finished practicing these database concepts, head on over to *Chapter 14*, *Serving Web Content*, where we'll journey into the world of web hosting.

## Further reading

- Configuring MariaDB with Option Files: `https://mariadb.com/kb/en/configuring-mariadb-with-option-files/`

# 14
# Serving Web Content

The flexible nature of Ubuntu Server makes it an amazing platform on which to host your organization's web presence. In this chapter, we'll take a look at Apache and NGINX, which make up the leading web server software on the internet. We'll go through installing, configuring, and extending both, as well as securing them with TLS. In addition, we'll also take a look at installing Nextcloud, which is a great solution for setting up your very own cloud environment for your organization to use for collaboration and sharing files. As we work through concepts related to hosting web content on Ubuntu Server, we will cover:

- Installing and configuring Apache
- Installing additional Apache modules
- Securing Apache with TLS
- Installing and configuring NGINX
- Setting up failover with `keepalived`
- Setting up and configuring Nextcloud

To get us started, we'll first look at configuring Apache, as well as some basic configuration.

## Installing and configuring Apache

The best way to become familiar with any technology is to dive right in. We'll begin this chapter by installing Apache. But first, what exactly is Apache? For those that aren't already aware, Apache is a popular application that is typically run on Linux and Unix servers to serve web pages to users. It runs in the background, and serves HTML pages to those that request a URL that exists on your server.

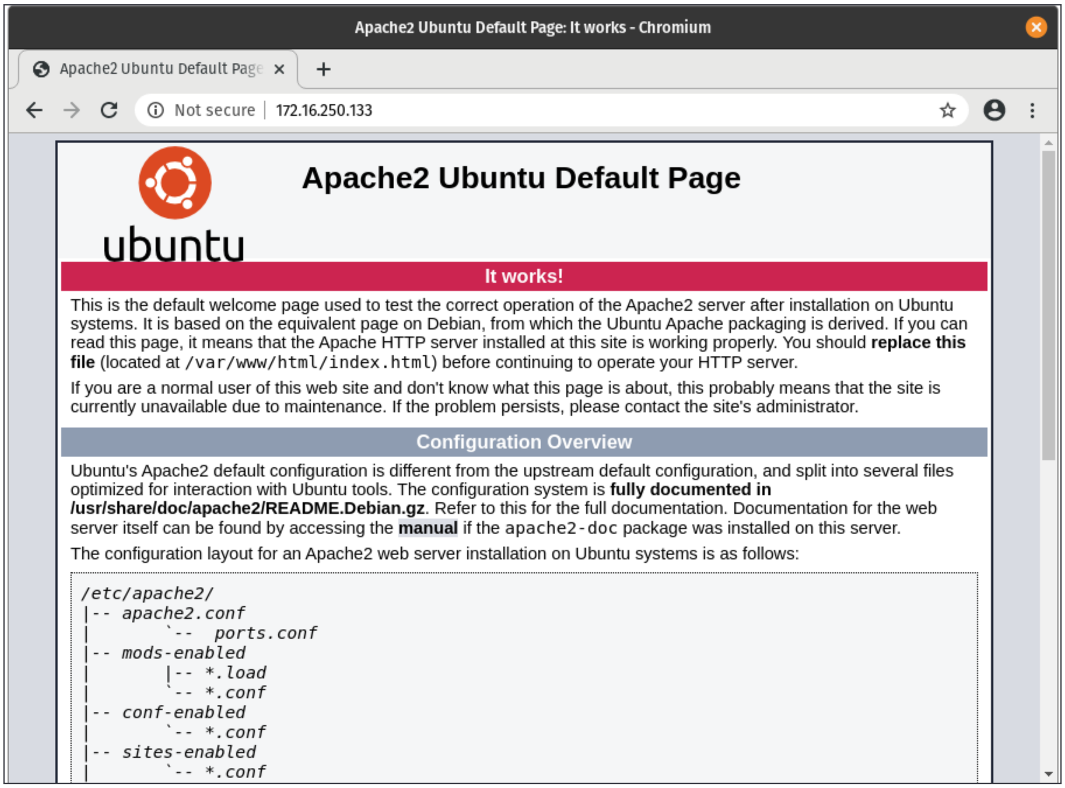
Installing Apache is very easy; it's simply a matter of installing the `apache2` package:

```
sudo apt install apache2
```

By default, Ubuntu will immediately start and enable the `apache2` daemon as soon as its package is installed. You can confirm this yourself with the following command:

```
systemctl status apache2
```

In fact, at this point, you already have (for all intents and purposes) a fully functional web server. If you were to open a web browser and enter the IP address of the server you just installed Apache on, you should see Apache's sample web page:


Apache2 Ubuntu Default Page: It works - Chromium
Apache2 Ubuntu Default Page
Not secure
172.16.250.133
ubuntu

Apache2 Ubuntu Default Page

It works!

This is the default welcome page used to test the correct operation of the Apache2 server after installation on Ubuntu systems. It is based on the equivalent page on Debian, from which the Ubuntu Apache packaging is derived. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should **replace this file** (located at `/var/www/html/index.html`) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

Configuration Overview

Ubuntu's Apache2 default configuration is different from the upstream default configuration, and split into several files optimized for interaction with Ubuntu tools. The configuration system is **fully documented in /usr/share/doc/apache2/README.Debian.gz**. Refer to this for the full documentation. Documentation for the web server itself can be found by accessing the **manual** if the `apache2-doc` package was installed on this server.

The configuration layout for an Apache2 web server installation on Ubuntu systems is as follows:

```
/etc/apache2/
|-- apache2.conf
|       `--  ports.conf
|-- mods-enabled
|       |-- *.load
|       `-- *.conf
|-- conf-enabled
|       `-- *.conf
|-- sites-enabled
|       `-- *.conf
```

Figure 14.1: The default sample web page provided by Apache

There you go, you have officially served web content. All you needed to do was install the `apache2` package, and your server was transformed into a web server. Chapter over, time to move on.

Of course, there's more to Apache than simply installing it and having it present a sample web page. While you could certainly replace the content in the sample web page with your own and be all set when it comes to hosting content to your users, there's much more to understand. For instance, there are several configuration files in the `/etc/apache2` directory that govern how sites are hosted, as well as which directories Apache will look in to find web pages to host. Apache also features plugins, which we will go over as well.

The directory that Apache serves web pages from is known as the **document root**, with `/var/www/html` being the default. Inside that directory, you'll see an `index.html` file, which is actually the default page you see when you visit an unmodified Apache server. Essentially, this is a test page that is designed to show you that the server is working, as well as some tidbits of information regarding the default configuration.

You're not limited to hosting just one website on a server, though. Apache supports the concept of a **virtual host**, which allows you to serve multiple websites from a single server. Each virtual host consists of an individual configuration file, which differentiates itself based on either name or IP address. For example, you could have an Apache server with a single IP address that hosts two different websites, such as `acmeconsulting.com` and `acmesales.com`. These are hypothetical websites, but you get the idea. To set this up, you would create separate configuration files for `acmeconsulting.com` and `acmesales.com` and store them in your Apache configuration directory. Each configuration file would include a `<VirtualHost>` stanza, where you would place an identifier such as a name or IP address that differentiates one from the other. When a request comes in, Apache will serve either `acmeconsulting.com` or `acmesales.com` to the user's browser, depending on which criteria matched when the request came in. The configuration files for each site typically end with the `.conf` filename extension, and are stored in the `/etc/apache2/sites-available` directory. We'll go over all of this in more detail shortly.

The basic workflow for setting up a new site (virtual host) will typically be similar to the following:

- The web developer creates the website and related files
- These files are uploaded to Ubuntu Server, typically in a sub-directory of `/var/www` or another directory the administrator has chosen
- The server administrator creates a configuration file for the site, and copies it into the `/etc/apache2/sites-available` directory
- The administrator enables the site and reloads Apache

Enabling virtual hosts is handled a bit differently in Debian and Ubuntu than in other platforms. In fact, there are two specific commands to handle this purpose: `a2ensite` for enabling a site and `a2dissite` for disabling a site. You won't find these commands on distributions such as CentOS, for example. Configuration files for each site are stored in the `/etc/apache2/sites-available/` directory and we would use the `a2ensite` command to enable each configuration. Assuming a site with the URL `acmeconsulting.com` is to be hosted on our Ubuntu server, we would create the `/etc/apache2/sites-available/acmeconsulting.com.conf` configuration file, and enable the site with the following commands:

```
sudo a2ensite acmeconsulting.com.conf
sudo systemctl reload apache2
```

I'm not using absolute paths in my examples; as long as you've copied the configuration file to the correct place, the `a2ensite` and `a2dissite` commands will know where to find it.

If we wanted to disable the site for some reason, we would execute the `a2dissite` command against the site's configuration file:

```
sudo a2dissite acmeconsulting.com.conf
sudo systemctl reload apache2
```

If you're curious about how this works behind the scenes, when the `a2ensite` command is run against a configuration file, it basically creates a symbolic link to that file and stores it in the `/etc/apache2/sites-enabled` directory. When you run `a2dissite` to disable a site, this symbolic link is removed. Apache, by default, will use any configuration files it finds in the `/etc/apache2/sites-enabled` directory. After enabling or disabling a site, you'll need to refresh Apache's configuration, which is where the `reload` option comes in. This command won't restart Apache itself (so users who are using your existing sites won't be disturbed) but it does give Apache a chance to reload its configuration files. If you replace `reload` with `restart` in the preceding commands, Apache will perform a full restart. You should only need to do that if you're having an issue with Apache or enabling a new plugin, but in most cases the `reload` option is preferred on a production system.

The main configuration file for Apache is located at `/etc/apache2/apache2.conf`. Feel free to view the contents of this file; the comments contain a good overview of how Apache's configuration is laid out. The following lines in this file are of special interest:

```
# Include the virtual host configurations:
IncludeOptional sites-enabled/*.conf
```

As you can see, this is how Ubuntu has configured Apache to look for enabled sites in the `/etc/apache2/sites-enabled` directory. Any file stored there with the `.conf` file extension is read by Apache. If you wish, you could actually remove those lines and Apache would then behave as it does on other platforms, and the `a2ensite` and `a2dissite` commands would no longer have any purpose. However, it's best to keep the framework of Ubuntu's implementation intact, as separating the configuration files makes logical sense and helps simplify the configuration. This chapter will go along with the Ubuntu way of managing configuration.

An additional virtual host is not required if you're only hosting a single site. The contents of `/var/www/html` are served by the default virtual host if you make no changes to Apache's configuration. This is where the example site that ships with Apache comes from. If you only need to host one site, you could remove the default `index.html` file stored in this directory and replace it with the files required by your website. If you wish to test this for yourself, you can make a backup copy of the default `index.html` file and create a new one with some standard HTML. You should see the default page change to feature the content you just added to the file.

The `000-default.conf` file is special, in that it's basically the configuration file that controls the default Apache sample website. If you look at the contents of the `/etc/apache2/sites-available` and `/etc/apache2/sites-enabled` directories, you'll see the `000-default.conf` configuration file stored in `sites-available` and `symlinked` in `sites-enabled`. This shows you that, by default, this site was included with Apache, and its configuration file was enabled as soon as Apache was installed. For all intents and purposes, the `000-default.conf` configuration file is all you need if you only plan on hosting a single website on your server. The contents of this file are as follows, but I've stripped the comments out of the file in order to save space on this page:

```
<VirtualHost *:80>
    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

As you can see, this default virtual host is telling Apache to listen on port `80` for requests and to serve content from `/var/www/html` as soon as requests come in. The `<VirtualHost>` declaration at the beginning is listening to everything (the asterisk is a *wildcard*) on port `80`, so this is basically handling all web traffic that comes into the server from port `80`. The `ServerAdmin` clause specifies the email address that is displayed in any error messages shown if there is a problem with the site.

The DocumentRoot setting tells Apache which directory to look for in order to find files to serve for connections to this virtual host. /var/www/html is the default, but some administrators choose to customize this. This file also contains lines for where to send logging information. The **access log** contains information relating to HTTP requests that come in, and by default is stored in /var/log/access.log. The **error log** is stored at /var/log/error.log and contains information you can use whenever someone has trouble visiting your site. The ${APACHE_LOG_DIR} variable equates to /var/log by default, and this is set in the /etc/apache2/envvars file, in case for some reason you wish to change this (for example, you wish to use a custom logging directory).

If you wish to host another site on the same server by creating an additional virtual host, you can use the same framework as the original file, with some additional customizations. Virtual host files are stored in the /etc/apache2/sites-available directory, with a filename ending in .conf. Here's an example for a hypothetical website, acmeconsulting.com. A virtual host file such as this might be saved as /etc/apache2/sites-available/acmeconsulting.com.conf:

```
<VirtualHost 192.168.1.104:80>
    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/acmeconsulting

    ErrorLog ${APACHE_LOG_DIR}/acmeconsulting.com-error.log
    CustomLog ${APACHE_LOG_DIR}/acmeconsulting.com-access.log combined
</VirtualHost>
```

To save time, I'll generally copy another virtual host file, even the default one, and change it accordingly. In this particular example, I've emphasized some important differences. First, with this virtual host, I'm not listening for all connections coming in on port 80; instead, I'm specifically looking for incoming traffic going to IP address 192.168.1.104 on port 80. This works because this server has two network cards, and therefore two IP addresses. With virtual hosts, I'm able to serve a different website, depending on which IP address the request is coming in on.

Next, I set DocumentRoot to /var/www/acmeconsulting. Each virtual host should have its own individual DocumentRoot to keep each site separate from the others. On my servers, I will typically disable or remove the sample virtual host (the one that has the default DocumentRoot of /var/www/html). Instead, I use /var/www as a base directory, and each virtual host gets its own directory as a sub-directory of this base.

Another change I find useful is to give each virtual host its own log files. Normally, Apache will use `/var/log/apache2/error.log` and `/var/log/apache2/access.log` to store log entries for all sites. If you only have a single site on your server, that is fine. However, when you're serving multiple sites, I find it useful to give each site its own independent log files. That way, if you are having trouble with a particular site, you don't have to scroll through unrelated log entries to find what you're looking for when you're troubleshooting. In my example, I inserted the website name in the log filenames, so this virtual host is logging errors in the `/var/log/apache2/acmeconsulting.com-error.log` file, and the access log is being written to `/var/log/apache2/acmeconsulting.com-access.log`. These log files will be created for you automatically as soon as you reload Apache.

With a server that only has a single IP address, you can still set up multiple virtual hosts. Instead of differentiating virtual hosts by IP, you can instead differentiate them by name. This is common on **Virtual Private Server** (**VPS**) installations of Ubuntu, where you'll typically have a single IP address assigned to you by your VPS provider. For name-based virtual hosts, we would use the `ServerName` option in our configuration. Refer to the following example to see how this would work. With this example, I'm adding name-based virtual hosts to their own files. I called mine `000-virtual-hosts.conf` and stored it in the directory. The contents are as follows:

```
<VirtualHost *:80>
    ServerName acmeconsulting.com
    DocumentRoot /var/www/acmeconsulting
</VirtualHost>

<VirtualHost *:80>
    ServerName acmesales.com
    DocumentRoot /var/www/acmesales
</VirtualHost>
```

For each virtual host, I'm declaring a `ServerName` with a matching `DocumentRoot`. With the first example, any traffic coming into the server requesting `acmeconsulting.com` will be provided a `DocumentRoot` of `/var/www/acmeconsulting`. The second example looks for traffic from `acmesales.com` and directs it to `/var/www/acmesales`. You can list as many virtual hosts here as you'd like to host on your server. Providing your server has enough resources to handle traffic to each site, you can host as many as you need.

If you're using domain names with virtual hosts, then this will only work if you set up networking such that the domain name referenced in the file resolves to the IP address of your server. Depending on your configuration, there are multiple ways to do this. If you're using a VPS provider, such as DigitalOcean or Linode, your server will have an IP address already and you only need to edit the **A Record** on your DNS server to point to that IP. (The various types of DNS entries, such as an A Record, were covered in *Chapter 11*, *Setting Up Network Services*.) If you're running your own DNS server, you would add the A Record there. If you're using an external DNS provider, you would log in to the dashboard for your account and add the A Record there. For testing purposes, you can edit your `/etc/hosts` file to point to your new web server. If you're not using a VPS provider, you would need to forward port `80` in your firewall to point to your internal web server. This is beyond the scope of this book, as there are many different models of firewalls available and it's impossible to cover them all.

As we continue through this chapter, we'll perform some additional configuration for Apache. At this point though, you should have an understanding of the basics of how Apache is configured in Ubuntu Server. For extra practice, feel free to create additional virtual hosts and serve different pages for them.

## Installing additional Apache modules

Apache features additional modules that can be installed that will extend its functionality. These modules can provide additional features such as adding support for things like Python or PHP. Ubuntu's implementation of Apache includes two specific commands for enabling and disabling modules, `a2enmod` and `a2dismod`, respectively. Apache modules are generally installed via packages from Ubuntu's repositories. To see a list of modules available for Apache, run the following command:

```
apt search libapache2-mod
```

In the results, you'll see various module packages available, such as `libapache2-mod-python` (which adds Python support) and `libapache2-mod-php7.4` (which adds PHP 7.4 support), among many others. Installing an Apache module is done the same way as any other package, with the `apt install` command. In the case of PHP support, we can install the required package with the following command:

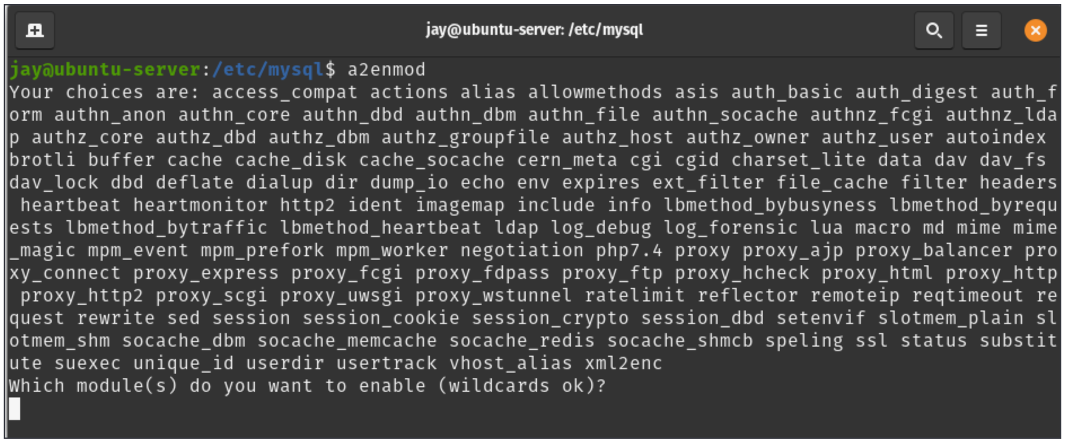
```
sudo apt install libapache2-mod-php7.4
```

Installing a module package alone is not enough for a module to be usable in Apache, though. Modules must be enabled in order for Apache to be able to utilize them. As mentioned earlier, we can use the `a2enmod` and `a2dismod` commands for enabling or disabling a module. You can view a list of modules that are built-in to Apache with the following command:

```
apache2 -l
```

The modules shown in the output will be those that are built into Apache, so you won't need to enable them. If the module your website requires is listed in the output, you're all set.

To view a list of all modules that are installed and ready to be enabled, you can run the `a2enmod` command by itself with no options:



```
jay@ubuntu-server:/etc/mysql$ a2enmod
Your choices are: access_compat actions alias allowmethods asis auth_basic auth_digest auth_f
orm authn_anon authn_core authn_dbd authn_dbm authn_file authn_socache authnz_fcgi authnz_lda
p authz_core authz_dbd authz_dbm authz_groupfile authz_host authz_owner authz_user autoindex
brotli buffer cache cache_disk cache_socache cern_meta cgi cgid charset_lite data dav dav_fs
dav_lock dbd deflate dialup dir dump_io echo env expires ext_filter file_cache filter headers
 heartbeat heartmonitor http2 ident imagemap include info lbmethod_bybusyness lbmethod_byrequ
ests lbmethod_bytraffic lbmethod_heartbeat ldap log_debug log_forensic lua macro md mime mime
_magic mpm_event mpm_prefork mpm_worker negotiation php7.4 proxy proxy_ajp proxy_balancer pro
xy_connect proxy_express proxy_fcgi proxy_fdpass proxy_ftp proxy_hcheck proxy_html proxy_http
 proxy_http2 proxy_scgi proxy_uwsgi proxy_wstunnel ratelimit reflector remoteip reqtimeout re
quest rewrite sed session session_cookie session_crypto session_dbd setenvif slotmem_plain sl
otmem_shm socache_dbm socache_memcache socache_redis socache_shmcb speling ssl status substit
ute suexec unique_id userdir usertrack vhost_alias xml2enc
Which module(s) do you want to enable (wildcards ok)?
```

Figure 14.2: The a2enmod command showing a list of available Apache modules

The end of the output of the `a2enmod` command will ask you whether or not you'd like to enable any of the modules:

```
Which module(s) do you want to enable (wildcards ok)?
```

If you wanted to, you could type the names of any additional modules you'd like to enable and then press *Enter*. Alternatively, you can press *Enter* without typing anything to simply return to the prompt.

If you give the `a2enmod` command a module name as an option, it will enable it for you. To enable PHP 7 (which we'll need later), you can run the following command:

```
sudo a2enmod php7.4
```

Chances are, though, if you've installed a package for an additional module, it was most likely was enabled for you during installation. With Debian and Ubuntu, it's very common for daemons and modules to be enabled as soon as their packages are installed, and Apache is no exception. In the case of the `libapache2-mod-php7.4` package I used as an example, the module should've been enabled for you once the package was installed:

```
sudo a2enmod php7.4
```

If a module is already enabled, you will see output similar to the following when you try to enable it with `a2enmod`:

```
Module php7.4 already enabled
```

If the module wasn't already enabled, we would see the following output:

```
Enabling module php7.4.
To activate the new configuration, you need to run:
systemctl restart apache2
```

As instructed, we'll need to restart Apache in order for the enabling of a module to take effect. Keep in mind, restarting Apache will make any sites it may host become unavailable during the process. When it comes to disabling a module, the command syntax is fairly similar. To do so, you'll use the `a2dismod` command along with the name of the module you'd like to disable:

```
sudo a2dismod php7.4
```

Enabling a module that was already previously enabled will result in output similar to the following:

```
Module php7.4 disabled.
```

To activate the new configuration, you need to run:

```
systemctl restart apache2
```

The modules you install and enable on your Apache server will depend on the needs of your website. For example, if you're going to need support for Python, you'll want to install the `libapache2-mod-python` package. If you're installing a third-party package, such as WordPress or Drupal, you'll want to refer to the documentation for those packages in order to obtain a list of which modules are required for the solution to install and run properly. Once you have such a list, you'll know which packages you'll need to install and which modules to enable.

## Securing Apache with TLS

Nowadays, it's a great idea to ensure your organization's website is encrypted and available over HTTPS. Encryption of web traffic has been historically achieved by utilizing **Secure Sockets Layer** (**SSL**), or more recently **Transport Layer Security** (**TLS**), which is the successor to SSL. Both refer to a method of utilizing cryptography by installing signed certificates that protect and encrypt web traffic. The two functions are different, but the end result is the same. Going forward, it's recommended to use TLS due to the additional security strength it offers, though it's not uncommon to see SSL being used nowadays since it hasn't been completely phased out.

Setting up and benefiting from TLS is not all that difficult to do, and will help protect your organization against common vulnerabilities being potentially exploited. Utilizing TLS doesn't protect you from all exploits being used in the wild, but it does offer a layer of protection you'll want to benefit from. Not only that, but your customers pretty much expect you to secure their communications nowadays. In this section, we'll look at how to use TLS with our Apache installation. We'll work through enabling it, generating certificates, and configuring Apache to use those certificates with both a single site configuration and with virtual hosts.

By default, Ubuntu's Apache configuration listens for traffic on port `80`, but not port `443` (HTTPS). You can check this yourself by running the following command:

```
sudo ss -tulpn | grep apache
```

The results will look similar to the following and will show the ports that Apache is listening on, which is only port `80` by default:

```
tcp   LISTEN 0      511                          *:80               *:*
users:(("apache2",pid=33019,fd=4),("apache2",pid=33018,fd=4),("apache2"
,pid=33017,fd=4),("apache2",pid=33016,fd=4),("apache2",pid=33015,fd=4),
("apache2",pid=33011,fd=4))
```

If the server were listening on port `443` as well, we would've seen the following within the output:

```
tcp   LISTEN 0      511                          *:443               *:*
```

To enable support for HTTPS traffic, we need to first enable the `ssl` module:

```
sudo a2enmod ssl
```

Next, we need to restart Apache:

```
sudo systemctl restart apache2
```

In addition to the sample website we discussed earlier, Ubuntu's default Apache implementation also includes another site configuration file, `/etc/apache2/sites-available/default-ssl.conf`. Unlike the sample site, this one is not enabled by default. This configuration file is similar to the sample site configuration but it's listening for connections on port `443` and contains additional configuration items related to TLS. Here's the content of that file, with the comments stripped out in order to save space on this page:

```
<IfModule mod_ssl.c>
        <VirtualHost _default_:443>
                ServerAdmin webmaster@localhost

                DocumentRoot /var/www/html

                ErrorLog ${APACHE_LOG_DIR}/error.log
                CustomLog ${APACHE_LOG_DIR}/access.log combined

                SSLEngine on

                SSLCertificateFile      /etc/ssl/certs/ssl-cert-
                snakeoil.pem
                SSLCertificateKeyFile /etc/ssl/private/ssl-cert-
                snakeoil.key

                <FilesMatch ".(cgi|shtml|phtml|php)$">
                                SSLOptions +StdEnvVars
                </FilesMatch>
                <Directory /usr/lib/cgi-bin>
                                SSLOptions +StdEnvVars
                </Directory>

        </VirtualHost>
</IfModule>
```

We already went over the `ServerAdmin`, `DocumentRoot`, `ErrorLog`, and `CustomLog` options earlier in this chapter, but there are additional options in this file that we haven't seen yet. On the first line, we can see that this virtual host is listening on port `443`. We also see `_default_` listed here instead of an IP address. The `_default_` option only applies to unspecified traffic, which in this case means any traffic coming into port `443` that hasn't been identified in any other virtual host. In addition, the `SSLEngine on` option enables TLS traffic. Right after that, we have options for our TLS certificate file and key file, which we'll get to a bit later.

We also have a `<Directory>` clause, which allows us to apply specific options to a directory. In this case, the `/usr/lib/cgi-bin` directory is having the `SSLOptions +StdEnvVars` settings applied, which enables default environment variables for use with TLS. This option is also applied to files that have an extension of `.cgi`, `.shtml`, `.phtml`, or `.php` through the `<FilesMatch>` option. The `BrowserMatch` option allows you to set options for specific browsers, though it's out of scope for this chapter. For now, just keep in mind that if you want to apply settings to specific browsers, you can.

By default, the `default-ssl.conf` file is not enabled. In order to benefit from its configuration options, we'll need to enable it, which we can do with the `a2ensite` command as we would with any other virtual host:

```
sudo a2ensite default-ssl.conf
```

Even though we just enabled TLS, our site isn't secure just yet. We'll need TLS certificates installed in order to secure our web server. We can do this in one of two ways, with self-signed certificates, or certificates signed by a certificate authority. Both are implemented in very similar ways, and I'll discuss both methods. For the purposes of testing, self-signed certificates are fine. In production, self-signed certificates would technically work, but most browsers won't trust them by default, and will give you an error when you go to their page. Therefore, it's a good idea to refrain from using self-signed certificates on a production system. Users of a site with self-signed certificates would need to bypass an error page before continuing to the site and seeing this error may cause them to avoid your site altogether. You can install the certificates into each user's web browser, but that can be a headache. In production, it's best to use certificates signed by a vendor.

Another method of setting up a certificate on your server is **Let's Encrypt**, a popular (and free) service for encrypting web traffic. Consider checking out the instructions at the Let's Encrypt website at `letsencrypt.org/docs`, as well as the example article mentioned at the end of the chapter.

As we go through this process, I'll first walk you through setting up TLS with a self-signed certificate so you can see how the process works. We'll create the certificate, and then install it into Apache. You won't necessarily need to create a website to go through this process, since you could just secure the sample website that comes with Apache if you wanted something to use as a proof of concept. After we complete the process, we'll take a look at installing certificates that were signed by a certificate authority.

To get the ball rolling, we'll need a directory to house our certificates. I'll use `/etc/apache2/certs` in my examples, although you can use whatever directory you'd like, as long as you remember to update Apache's configuration with your desired location and filenames:

```
sudo mkdir /etc/apache2/certs
```

For a self-signed certificate and key, we can generate the pair with the following command. Feel free to change the name of the key and certificate files to match the name of your website:

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout \ /
etc/apache2/certs/mysite.key -out /etc/apache2/certs/mysite.crt
```

You'll be prompted to enter some information for generating the certificate. Answer each prompt as they come along. Here's a list of the questions you'll be asked, along with my responses for each. Change the answers to fit your server, environment, organization name, and location:

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Michigan
Locality Name (eg, city) []:Detroit
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Company
Organizational Unit Name (eg, section) []:IT
Common Name (e.g. server FQDN or YOUR name) []:myserver.mydomain.com
Email Address []:webmaster@mycompany.com
```

Now, you should see that two files have been created in the `/etc/apache2/certs` directory, `mysite.crt` and `mysite.key`, which represent the certificate and private key, respectively. Now that these files have been generated, the next thing for us to do is to configure Apache to use them. Look for the following two lines in the `/etc/apache2/sites-available/default-ssl.conf` file:

```
SSLCertificateFile       /etc/ssl/certs/ssl-cert-snakeoil.pem
SSLCertificateKeyFile  /etc/ssl/private/ssl-cert-snakeoil.key
```

Comment these lines out by placing a `#` symbol in front of both:

```
# SSLCertificateFile      /etc/ssl/certs/ssl-cert-snakeoil.pem
# SSLCertificateKeyFile /etc/ssl/private/ssl-cert-snakeoil.key
```

Next, add the following two lines underneath the lines you just commented out. Be sure to replace the target directories and certificate filenames with yours, if you followed your own naming convention:

```
SSLCertificateFile      /etc/apache2/certs/mysite.crt
SSLCertificateKeyFile /etc/apache2/certs/mysite.key
```

To make Apache benefit from the new configuration, reload the `apache2` daemon:

```
sudo systemctl reload apache2
```

With the new configuration in place, we're not quite done but we're close. We still have a small bit of configuration left to add. But before we get to that, let's return to the topic of installing TLS certificates that were signed by a certificate authority. The process for installing signed TLS certificates is pretty much the same, but the main difference is how the certificate files are requested and obtained. Once you have them, you will copy them to your file server and configure Apache the same way as we just did. To start the process of obtaining a signed TLS certificate, you'll need to create a **Certificate Signing Request** (**CSR**). A CSR is basically a request for a certificate in file form that you'll supply to your certificate authority to start the process of requesting a signed certificate. A CSR can be easily generated with the following command:

```
openssl req -new -newkey rsa:2048 -nodes -keyout server.key -out
server.csr
```

With the CSR file that was generated, you can request a signed certificate. The CSR file should now be in your current working directory. The entire process differs from one provider to another, but in most cases, it's fairly straightforward. You'll send them the CSR, pay their fee, fill out a form or two on their website, prove that you are the owner of the website in question, and then the vendor will send you the files you need. It may sound complicated, but certificate authorities usually walk you through the entire process and make it clear what they need from you in order to proceed. Once you complete the process, the certificate authority will send you your certificate files, which you'll then install on your server. Once you configure the `SSLCertificateFile` and `SSLCertificateKeyFile` options in `/etc/apache2/sites-available/default-ssl.conf` to point to the new certificate files and reload Apache, you should be good to go.

There's one more additional step we should perform for setting this up properly. At this point, our certificate files should be properly installed, but we'll need to inform Apache of when to apply them. If you recall, the `default-ssl.conf` file provided by the `apache2` package is answering requests for any traffic not otherwise identified by a virtual host (the `<VirtualHost _default_:443>` option). We will need to ensure our web server is handling traffic for our existing websites when TLS is requested. We can add a `ServerName` option to that file to ensure our site supports TLS.

Add the following option to the `/etc/apache2/sites-available/default-ssl.conf` file, right underneath `<VirtualHost _default_:443>`:

```
ServerName mydomain.com:443
```

Now, when traffic comes into your server on port `443` requesting a domain that matches the domain you typed for the `ServerName` option, it should result in a secure browsing session for the client. You should see the green padlock icon in the address bar (this depends on your browser), which indicates your session is secured. If you're using self-signed certificates, you'll probably see an error you'll have to skip through first, and you may not get the green padlock icon. This doesn't mean the encryption isn't working; it just means your browser is skeptical of the certificate since it wasn't signed by a known certificate authority. Your session will still be encrypted.

If you are planning on hosting multiple websites over HTTPS, you may want to consider using a separate virtual host file for each. An easy way to accomplish this is to use the `/etc/apache2/sites-available/default-ssl.conf` file as a template and change `DocumentRoot` to the directory that hosts the files for that site. In addition, be sure to update the `SSLCertificateFile` and `SSLCertificateKeyFile` options to point to the certificate files for the site and set `ServerName` to the domain that corresponds to your site. Here's an example virtual host file for a hypothetical site that uses TLS. I've highlighted lines that I've changed from the normal `default-ssl.conf` file:

```
<IfModule mod_ssl.c>
        <VirtualHost *:443>
                ServerName acmeconsulting.com:443

                ServerAdmin webmaster@localhost

                DocumentRoot /var/www/acmeconsulting
                ErrorLog ${APACHE_LOG_DIR}/acmeconsulting.com-error.log
                CustomLog ${APACHE_LOG_DIR}/acmeconsulting.com-access.
                log combined

                SSLEngine on
        SSLCertificateFile      /etc/apache2/certs/acmeconsulting/acme.
        crt
        SSLCertificateKeyFile /etc/apache2/certs/acmeconsulting/acme.
        key

                <FilesMatch ".(cgi|shtml|phtml|php)$">
                                SSLOptions +StdEnvVars
                </FilesMatch>
                <Directory /usr/lib/cgi-bin>
                                SSLOptions +StdEnvVars
                </Directory>

        </VirtualHost>
</IfModule>
```

Basically, what I did was create a new virtual host configuration file (using the existing `default-ssl.conf` file as a template). I called this new file `acme-consulting.conf` and I stored it in the `/etc/apache2/sites-available` directory. I changed the `VirtualHost` line to listen for anything coming in on port `443`. The line `ServerName acmeconsulting.com:443` was added to make this file responsible for traffic coming in looking for `acmeconsulting.com` on port `443`. I also set `DocumentRoot` to `/var/www/acmeconsulting`. In addition, I customized the error and access logs so that it will be easier to find log messages relating to this new site, since its log entries will go to their own specific files.

In my experience, I find that a modular approach, such as what I've done with the sample virtual host file for HTTPS, works best when setting up a web server that's intended to host multiple websites. With each site, I'll typically give it its own document root, certificate files, and log files. Even if you're only planning on hosting a single site on your server, using this modular approach is still a good idea, since you may want to host additional sites later on.

So, there you have it. You should now understand how to set up secure virtual hosts in Apache. However, security doesn't equal redundancy.

## Installing and configuring NGINX

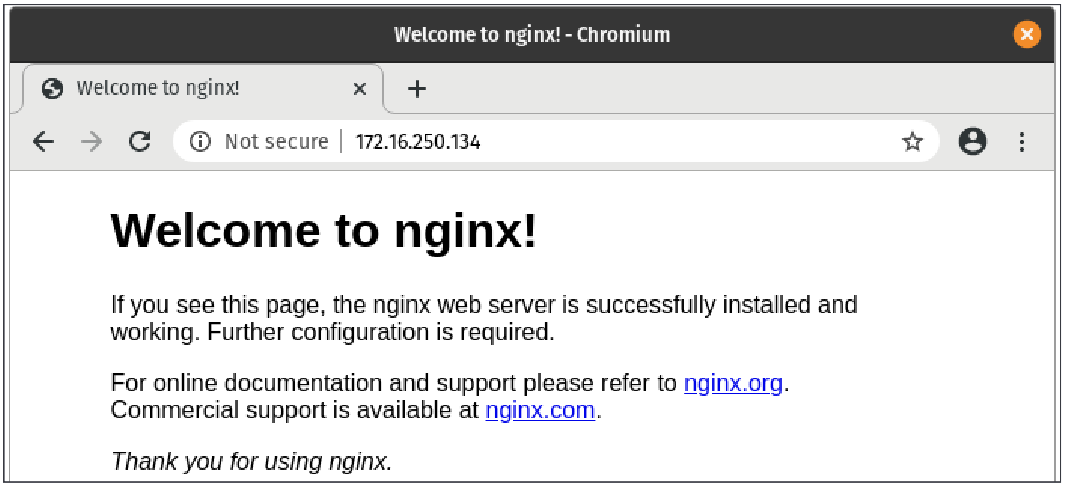
Apache isn't the only technology that is capable of allowing you to host web content on your server. NGINX also serves the same purpose and is gaining popularity quite rapidly. Although Apache is still the most common option right now, it's a good idea to at least be familiar with NGINX and learn its basics. NGINX itself is a proxy server as well, but is capable of also serving web content, which is why it competes with Apache. Serving web content is our focus for covering it in this section.

Before we do so, I want to mention first that you can really only have one web server service running on a single web server. If you've been following along up to now, you currently have a functional Apache web server. If you were to also install NGINX, it probably wouldn't start as the ports it wants to listen on (port `80` and/or `443`) will already be in use. You can run both on a single server, but that's outside the scope of this book. Ideally, you'd want to use one or the other. Therefore, to continue with this section you'd either want to remove Apache or set up a separate web server for testing NGINX. I recommend the latter, because later on in this chapter we will take a look at hosting Nextcloud and we will be using Apache to do so. If you remove Apache now, you'd have to add it back in order to follow along with that section. Theoretically, you'd only have to stop the `apache2` process before starting `nginx`, but the two resources sharing the same server has a lot of variables and may conflict.

To get started with NGINX, simply install it:

```
sudo apt install nginx
```

Just like with Apache, if we enter the IP address of our server in a browser, we're presented with a sample page, but this time NGINX's version instead of the one that ships with Apache. It certainly looks boring in comparison, but it works:



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

*Thank you for using nginx.*

Figure 14.3: The NGINX sample page

The default configuration files for `nginx` are stored in the `/etc/nginx` directory. Go ahead and peruse these files to get a general feel for how the configuration is presented. Similar to Apache, you also have a `sites-enabled` and `sites-available` directory here, which serve the same purpose.

Just as with Apache, the `sites-available` directory houses configuration files for sites that *can be* enabled, while the `sites-enabled` directory stores configuration files for sites that are enabled. Unlike Apache, though, we don't have dedicated commands to enable these sites. We have to link them manually. Although we haven't even looked at NGINX configuration files yet, let's just assume that we have created the following configuration file:

```
/etc/nginx/sites-available/acmesales.com
```

To enable that site, we would need to create a symbolic link for it and store that link in the `/etc/nginx/sites-enabled` directory:

```
sudo ln -s /etc/nginx/sites-available/acmesales.com /etc/nginx/sites-
enabled/acmesales.com
```

Then, we can reload `nginx`:

```
sudo systemctl reload nginx
```

As it stands right now, a site configuration file named `default` exists in `/etc/nginx/sites-available` and a symbolic link to it is already present in `/etc/nginx/sites-enabled`. If all we want to do is host a single site, we only need to replace the default content that NGINX serves, which is located in the `/var/www/html` directory (the same as Apache) with the content for our site. After refreshing the page, we're good to go.

If we want to serve more than one site from one server, the `default` file is a great starting point for creating additional virtual hosts. We can start by copying it to a new name:

```
sudo cp /etc/nginx/sites-available/default /etc/nginx/sites-available/
acmesales.com
```

Obviously, `acmesales.com` is an example, so feel free to name this whatever you wish.

Now, we can edit this file and change it to serve additional content. First of all, only one site can be referred to as a *default* site. A default site in NGINX is one that answers if none of the other sites match a request. Therefore, we want to remove both occurrences of `default_server` from our newly copied config. Find these lines:

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

Change them to this:

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

Next, we'll need to adjust the `server_name` option to refer to the name of our new site. Add this line:

```
server_name acmesales.com www.acmesales.com;
```

Now, we'll need to change the document root to the directory that will store the files for our new site. Find this line:

```
root /var/www/html;
```

And change it to this:

```
root /var/www/acmesales.com;
```

The final file should look like the following at this point:

```
server {
        listen 80;
        listen [::]:80;

        root /var/www/acmesales.com;

        index index.html index.htm index.nginx-debian.html;

        server_name acmesales.com www.acmesales.com;

        location / {
                try_files $uri $uri/ =404;
        }
}
```

You can probably see that the configuration format for NGINX configuration files is simpler than with Apache. I find this to be true, and I've noticed that sites I've configured with NGINX generally have fewer lines in their configuration files than Apache does.

At this point, assuming that you have the required content in `/var/www/acmesales.com` and have a proper configuration file, the new site should respond as soon as you reload `nginx`. But what about TLS? I recommend we always secure our websites, regardless of which solution we're using to serve it. With NGINX, we can add that feature easily. The certificate files themselves are the same regardless of whether we're using Apache or NGINX. If you haven't already created your certificate files, refer back to the section in this chapter where we did so. Assuming you already have certificate files, we just need to make additional changes to our configuration.

First, we change the first two lines to listen on port `443` with TLS instead of standard port `80`:

```
listen 443 ssl;
listen [::]:443 ssl;
```

Next, we'll add the following two lines before the `location` section:

```
ssl_certificate /etc/certs/cert.pem;
ssl_certificate_key /etc/certs/cert.key;
ssl_session_timeout 5m;
```

For this to work, you'll need to adjust the paths and the names of the `cert` files to make sure they match what you called them on your server. The entire file should look similar to the following at this point:

```
server {
        listen 443 ssl;
        listen [::]:443 ssl;

        root /var/www/html;

        index index.html index.htm index.nginx-debian.html;

        server_name acmesales.com www.acmesales.com;

         ssl_certificate /etc/certs/cert.pem;
         ssl_certificate_key /etc/certs/cert.key;
         ssl_session_timeout 5m;
        location / {
                try_files $uri $uri/ =404;
        }
}
```

Finally, a potential problem is that users may access our site via port `80`, instead of utilizing HTTPS. We can tell NGINX to forward these people to the secure version of our site automatically. To do that, we can edit the default configuration file (`/etc/nginx/sites-available/default`) and add the following line just after the two `listen` directives:

```
return 301 https://$host$request_uri;
```

Now, anytime a user visits the HTTP version of our site, they'll be redirected to the secure HTTPS version automatically.

Now that we've looked at serving web content with both Apache and NGINX, let's take a moment to discuss **keepalived**, which can help us achieve high availability as well.

## Setting up failover with keepalived

When we host important sites or applications, especially those that may be critical to conducting business, high availability is a valuable addition to increase stability. In short, if we have a single server that serves a single resource, then we will experience downtime if something were to happen to that server. We may be able to recover from this situation fairly quickly by using a template, image, or some other backup and restore methodology to get back to a working state, but it's even faster to recover from such a situation if we are able to direct our users to an alternate server while we fix the primary, or implement some sort of logic to enable us to route users away from a problematic server.

There are many ways a situation like this can be handled, and `keepalived` is one way we can see this in action quickly, as setting it up is fairly straightforward. Before we continue though, note that implementing a high availability or similar solution manually only makes sense for virtual servers or physical infrastructure you maintain yourself, as many cloud providers have their own solution you can use for the same purpose. Therefore, this section has the most value to those that manage and maintain their own network rather than using a cloud platform.

Using `keepalived` is a great way to add high availability to an application or even a hosted website. `keepalived` allows you to configure a floating IP (also known as a **Virtual IP** or **VIP**) for a pool of servers, with this special IP being applied to a single server at a time. Each installation of `keepalived` in the same group will be able to detect when another server isn't available, and claim ownership of the floating IP whenever the primary server isn't responding. This allows you to run a service on multiple servers, with a server taking over in the event that another becomes unavailable.

keepalived is by no means specific to Apache. You can use it with many different applications and services, NGINX being another example. keepalived also allows you to create a load-balanced environment as well.

Let's talk a little bit about how keepalived can work with Apache in theory. Once keepalived is installed on two or more servers, it can be configured such that a primary server will have ownership of the floating IP, and other servers will continually check the availability of the primary, claiming the floating IP for themselves whenever the current primary becomes unreachable. For this to work, each server would need to contain the same Apache configuration and site files. I'll leave it up to you to set up multiple Apache servers. If you've followed along so far, you should have at least one already. If it's a virtual machine, feel free to create a clone of it and use that for the secondary server. If not, all you should have to do is set up another server, following the instructions earlier in this chapter. If you're able to view a website on both servers, you're good to go.

To get started, we'll need to declare a floating IP. This can be any IP address that's not currently in use on your network. For this to work, you'll need to set aside an IP address for the purposes of keepalived. If you don't already have one, pick an IP address that's not being used by any device on your network. If you're at all unsure, you can use an IP scanner on your network to find an available IP address. There are several scanners that can accomplish this, such as nmap on Linux or **Angry IP Scanner** for Windows (I have no idea what made that IP scanner so angry, but it does work well).

Be careful when scanning networks for free IP addresses, as scanning a network may show up in intrusion detection systems as a threat. If you're scanning any network other than one that you own, always make sure you have permission from both the network administrator as well as management before you start scanning. Also, if you're scanning a company network, keep in mind that any hardware load balancers in use may not respond to pings, so you may want to also look at an IP layout from your network administrator as well and compare the results.

To save you the trouble of a Google search, the following nmap syntax will scan a subnet and report back regarding which IP addresses are in use. You'll need the nmap package installed first. Just replace the network address with yours:

```
nmap -sP 192.168.1.0/24
```

Next, we'll need to install `keepalived`. Simply run the following command on both of your servers:

```
sudo apt install keepalived
```

If you check the status of the `keepalived` daemon, you'll see that it attempted to start as soon as it was installed, and then immediately failed. If you check the status of `keepalived` with the `systemctl` command, you'll see an error similar to the following:

```
Condition: start condition failed
```

Since we haven't actually configured `keepalived`, it's fine for it to have failed. After all, we haven't given it anything to check, nor have we assigned our floating IP.
By default, there is no sample configuration file created once you've installed the package; we'll have to create that on our own. Configuration for `keepalived` is stored in `/etc/keepalived`, which at this point should just be an empty directory on your system. If, for some reason, this directory doesn't exist, create it:

```
sudo mkdir /etc/keepalived
```

Let's open a text editor on our primary server and edit the `/etc/keepalived/keepalived.conf` file, which should be empty. Insert the following code into the file. There are some changes you'll need to make in order for this configuration to work in your environment. As we continue, I'll explain what the code is doing and what you'll need to change. Keep your text editor open after you paste the code, and keep reading for some tips on how to ensure that this `config` file will work for you:

```
global_defs {
    notification_email {
    myemail@mycompany.com
    }
    notification_email_from keepalived@mycompany.com
    smtp_server 192.168.1.150
    smtp_connect_timeout 30
    router_id mycompany_web_prod
}
vrrp_instance VI_1 {
    smtp_alert
    interface enp0s3
    virtual_router_id 51
    priority 100
     advert_int 5
    virtual_ipaddress {
    192.168.1.200
    }
}
```

There's quite a bit going on in this file, so I'll explain the configuration section by section, so you can better understand what's happening:

```
global_defs {
    notification_email {
    myemail@mycompany.com
    }
    notification_email_from keepalived@mycompany.com
    smtp_server 192.168.1.150
    smtp_connect_timeout 30
    router_id mycompany_web_prod
}
```

In the `global_defs` section, we're specifically configuring an email host to use in order to send out alert messages. When there's an issue and `keepalived` switches to a new primary server, it can send you an email to let you know that there was a problem. You'll want to change each of these values to match that of your mail server. If you don't have a mail server, `keepalived` will still function properly. You can remove this section if you don't have a mail server, or comment it out. However, it's highly recommended that you set a mail server for `keepalived` to use.

Let's look at the next section of code:

```
vrrp_instance VI_1 {
    smtp_alert
    interface enp0s3
    virtual_router_id 51
    priority 100

    advert_int 5
```

Here, we're assigning some configuration options regarding how our virtual IP assignment will function. The first thing you'll want to change here is the interface. In my sample `config`, I have `enp0s3` as the interface name. This will need to match the interface on your server where you would like `keepalived` to be bound to. If in doubt, execute `ip a` at your shell prompt to get a list of interfaces.

`virtual_router_id` is a very special number that can be anything from `0` to `255`. This number is used to differentiate multiple instances of `keepalived` running on the same subnet. Each member of each `keepalived` cluster should have the same `virtual_router_id`, but if you're running multiple pairs or groups of servers on the same subnet, each group should have its own `virtual_router_id`. I used `51` in my example, but you can use whatever number you'd like.

You'll just have to make sure both web servers have the same `virtual_router_id`. I'll go over some tips for setting up the other server shortly, so don't worry if you haven't configured `keepalived` on your second server yet.

Another option in this block that's important is `priority`. This number must absolutely be different on each server. The `keepalived` instance with the highest `priority` value is always considered the primary server. In my example, I set this number to `100`. Using `100` for the priority is fine, so long as no other server is using that number. Other servers should have a lower priority. For example, you can set the second web server's priority to `80`. If you set up a third or fourth one, you could set their priority to `60` and `40`, respectively. Those are just arbitrary numbers I picked off the top of my head. As long as each server has a different priority, and the server you'd like to be the primary has the highest priority, you're in good shape. The `advert_int` option configures how frequently advertisements are sent, and is configured in seconds. In our example code, advertisements are sent every 5 seconds, which informs the other server that it's still alive and active.

In the final block, we're setting the virtual IP address:

```
    virtual_ipaddress {
    192.168.1.200
}
```

I chose `192.168.1.200` as a hypothetical example; you should choose an IP address outside of your DHCP range that's not being used by any device.

Now that you've configured `keepalived`, let's test it out and see whether it's working. On your primary server, start `keepalived`:

```
sudo systemctl start keepalived
```

After you start the daemon, take a look at its status to see how it's doing:

```
sudo systemctl status keepalived
```

If there are no errors, the status of the daemon should be `active (running)`. If for some reason the status is something else, take a look at the log entries shown after you execute the `status` command, as `keepalived` is fairly descriptive regarding any errors it finds in your `config` file.

If all went well, you should see the IP address you chose for the floating IP listed in your interfaces. Execute `ip a` at your prompt to see them. Do you see the floating IP? If not, keep in mind it can take a handful of seconds for it to show up. Wait 30 seconds or so and check your interface list again.

If all went well on the first server, we should set up `keepalived` on the second web server. Install the `keepalived` package on the other server if you haven't already done so, and then copy the `keepalived.conf` file from your working server to your new one. Here are some things to keep in mind regarding the second server's `keepalived.conf` file:

- Be sure to change the priority on the secondary server to a lower number than what you used on the primary server
- The `virtual_ipaddress` value should be the same on both
- The `virtual_router_id` value should be the same on both
- Start or restart `keepalived` on your secondary server, and verify there were no errors when the service started up

Now, let's have some fun and see `keepalived` in action. What follows is an extremely simple HTML file:

```
<html>
    <title>keepalived test</title>
    <body>
        <p>This is server #1!</p>
    </body>
</html>
```

Copy this same HTML file to both servers and serve it via Apache or NGINX. We've gone over setting up both in this chapter. The easiest and quickest way to implement this is to replace Apache's sample HTML file with the preceding one. Make sure you change the text `This is server #1!` to `This is server #2` on the second server. You should be able to use your browser and visit the IP address for each of your two web servers and see this page on both. The only difference between them should be the text on the page.

Now, in your browser, change the URL you're viewing to that of your floating IP. You should see the same page as you did when you visited the IP address for `server #1`.

Let's now simulate a failover. On the primary server, stop the `keepalived` service:

```
sudo systemctl stop keepalived
```

In our `config`, we set the `advert_int` option to `5`, so the second server should take over within 5 seconds of you stopping `keepalived` on the first server. When you refresh the page, you should see the web page for `server #2` instead of `server #1`. If you execute `ip a` on the shell to view a list of interfaces on your secondary server, you should also see that the secondary is now holding your floating IP address. Neat, isn't it? To see your primary reclaim this IP, start the `keepalived` daemon on your primary server and wait a few seconds.

Congratulations, you set up a cluster for your Apache implementation. If the primary server encounters an issue and drops off the network, your secondary server will take over. As soon as you fix your primary server, it will immediately take over hosting your web page. Although we made our site slightly different on each server, the idea is showing how hosting a website can survive a single server failure. You can certainly host more than just Apache with `keepalived`. Almost any service you need to be highly available that doesn't have its own clustering options is a good candidate for `keepalived`. For additional experimentation, try adding a third server.

## Setting up and configuring Nextcloud

I figured we'd end this chapter with a fun activity: setting up our very own Nextcloud server. Nextcloud is a very useful web application that's handy for any organization. Even if you're not working on a company network, Nextcloud is a great asset for even a single user. You can use it to synchronize files between machines, store and sync contacts, keep track of tasks you're working on, fetch email from a mail server, and more. To complete this activity, you'll need a web server to work with. Nextcloud supports multiple different web server platforms, but in this example, we'll be using Apache.

You'll also need an installation of MySQL or MariaDB, as Nextcloud will need its own database. We went over installing and managing MariaDB databases in *Chapter 13*, *Setting Up a Database Server*. I'll give you all the commands you'll need to set up the database in this section, but refer back to *Chapter 13, Setting Up a Database Server*, if any of these commands confuse you.

To get started, we need to download Nextcloud. To do so, head on over to the project's website at `https://www.nextcloud.com` and navigate to the **Download** section. The layout of this site may change from time to time, but as of the time of writing, the first link to click on is a button labeled **Get Nextcloud**, and once you highlight that button with your mouse or pointer, you will see a **Server Packages** link. Once there, look for the **Download Nextcloud** button, but don't click on it. Instead, right-click on it and click **Copy link address** or a similarly named option, depending on the browser you use.

This should copy the link for the download to your clipboard. If the Nextcloud website layout has changed since publication, you're essentially just looking for the URL to the ZIP file to download Nextcloud, and you'll want to copy it to your computer's clipboard.

Next, open an SSH session to your web server. Make sure you're currently working from your home directory, and execute the following command:

```
wget <URL of Nextcloud>
```

To get the Nextcloud URL, simply paste the URL into your terminal after typing `wget`. Your entire command will look similar to the following (the version number changes constantly as they release new versions):

```
wget https://download.nextcloud.com/server/releases/nextcloud-
19.0.1.zip
```

This command will download the Nextcloud software locally to your current working directory. Next, we'll need to `unzip` the archive:

```
unzip nextcloud-19.0.1.zip
```

If you get an error message insinuating that the `unzip` command is not available, you may need to install it:

```
sudo apt install unzip
```

Now, let's move the newly extracted `nextcloud` directory to `/var/www/html`:

```
sudo mv nextcloud /var/www/html/nextcloud
```

In order for Nextcloud to function, the user account that Apache uses to serve content will need full access to it. Let's use the following command to give the user `www-data` ownership of the `nextcloud` directory:

```
sudo chown www-data:www-data -R /var/www/html/nextcloud
```

Now, you should have the required files for the Nextcloud software installed on the server in the `/var/www/nextcloud` directory. In order for this to work, though, Apache will need a configuration file that includes `/var/www/nextcloud` as its document root. We can create the file we need at the following location:

```
/etc/apache2/sites-available/nextcloud.conf
```

Example content to include in that file is as follows:

```
Alias /nextcloud "/var/www/html/nextcloud/"

<Directory /var/www/html/nextcloud/>
  Options +FollowSymlinks
  AllowOverride All

 <IfModule mod_dav.c>
  Dav off
 </IfModule>

 SetEnv HOME /var/www/html/nextcloud
 SetEnv HTTP_HOME /var/www/html/nextcloud
</Directory>
```

Similar to our earlier discussion on Apache, we're adding a config file here specifically for Nextcloud that sets up an alias to point to `/nextcloud` to `www.mydomain.com/nextcloud`. Essentially, it allows Nextcloud to be reached at your domain name, plus `/nextcloud` added to the end. The rest of the file disables WebDav (a means of allowing a web server to act as a file server, not needed in our case) and then enables environment variables to set `HOME` and `HTTP_HOME` to the document root for Nextcloud, which in our case is `/var/www/html/nextcloud`.

Next, we enable the new site:

```
sudo a2ensite nextcloud.conf
```

Next, we'll need to make a change to Apache. First, we'll need to ensure that the `libapache2-mod-php7.4` package is installed since Nextcloud requires PHP, but we'll need some additional packages as well. You can install Nextcloud's prerequisite packages with the following command:

```
sudo apt install libapache2-mod-php7.4 php7.4-curl php7.4-gd php7.4-
intl php7.4-mbstring php7.4-mysql php7.4-xml php7.4-zip
```

Next, restart Apache so it can take advantage of the new PHP plugin:

```
sudo systemctl restart apache2
```

At this point, we'll need a MySQL or MariaDB database for Nextcloud to use. This database can exist on another server, or you can share it on the same server you installed Nextcloud on. If you haven't already set up MariaDB, a walk-through was covered during *Chapter 13*, *Managing Databases*. At this point, it's assumed that you already have MariaDB installed and running.

Log in to your MariaDB instance as your `root` user, or a user with full root privileges. You can create the Nextcloud database with the following command:

```
CREATE DATABASE nextcloud;
```

Next, we'll need to add a new user to MariaDB for Nextcloud and give that user full access to the `nextcloud` database. We can take care of both with the following command:

```
GRANT ALL ON nextcloud.* to 'nextcloud'@'localhost' IDENTIFIED BY
'super_secret_password';
```

Make sure to change `super_secret_password` to a very strong (preferably randomly generated) password. Make sure you save this password in a safe place.

Now we have all we need in order to configure Nextcloud. You should now be able to visit your Nextcloud instance in a web browser. Just enter a URL similar to the following, replacing the sample IP address with the one for your server:

```
http://172.16.250.133/nextcloud
```

If you're using a subdomain and gave Nextcloud its own virtual host, that URL would then be something like this:

```
http://nextcloud.yourdomain.com
```

You should see a page asking you to configure Nextcloud:

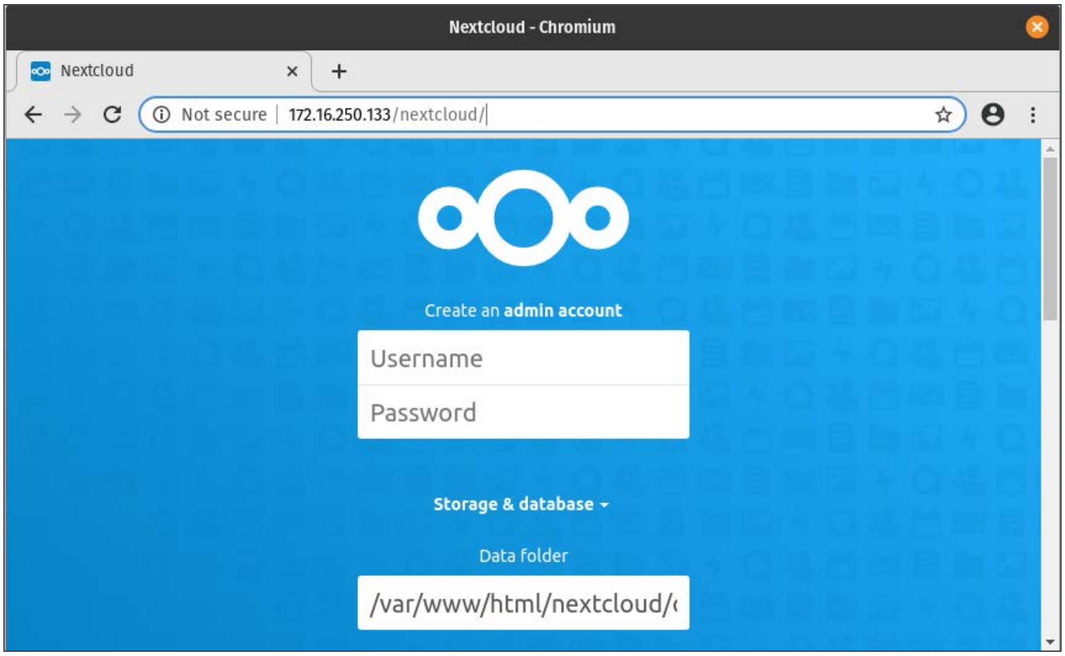


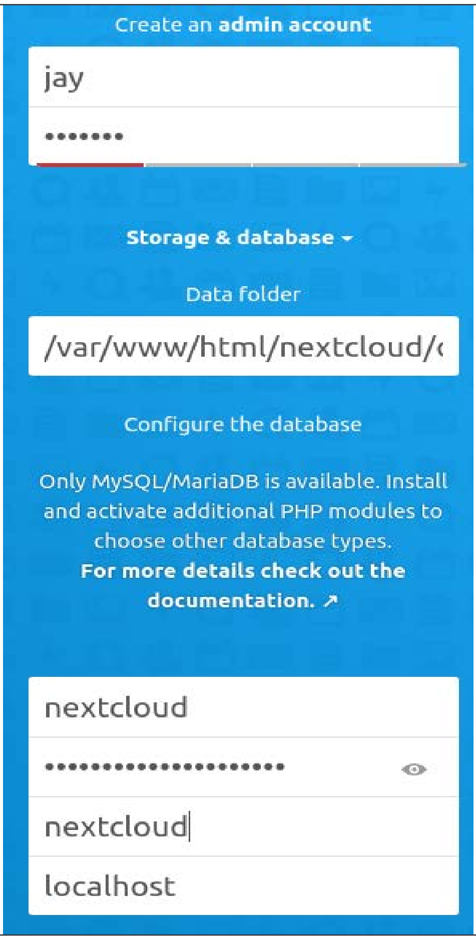
Figure 14.4: Nextcloud configuration page

If you do not see this page, make sure that the `/var/www/html/nextcloud` directory is accessible via Apache. Also, make sure you have an appropriate virtual host for Nextcloud referencing this directory as its document root.

This page will ask you for several pieces of information. First, you'll see **Username** and **Password**. This is not asking you for a pre-existing account, but actually to set up a brand-new administrator account. This shouldn't be an account you'll use on a day-to-day basis, but instead an admin account you'll use only when you want to add users and maintain your system. Please note that it won't ask you to confirm the password, so you'll want to make certain you're entering the password you think you are. It's perhaps safer to type the password in a text editor, and then copy and paste the password into the **Password** box to make sure you don't lock yourself out.

**Data folder** will default to `/var/www/html/nextcloud/data`. This default is normally fine, but if you have configured your server to have a separate data partition, you can configure that here. If you plan on storing a large amount of data on your Nextcloud server, setting up a separate partition for it may be a good idea. If you do, you can set that here. Otherwise, leave the default.

In the next section, you'll be asked to fill in information for the Nextcloud database we created earlier. **Database user** and **Database password** will use the values we created when we set up the MariaDB database for Nextcloud earlier. In my examples, I used `nextcloud` for the username as well as **Database name**. The password will be whatever it is you used for the password when we set up the database user account and granted privileges. Finally, the database server defaults to `localhost`, which is correct as long as you set up the database on the same machine as the Nextcloud server. If not, enter the address of your database server here, if it's somewhere else. The following screenshot shows the initial form completely filled out with the example values we've used so far in this section:



Create an **admin account**

jay

•••••••

**Storage & database**

Data folder

/var/www/html/nextcloud/d

Configure the database

Only MySQL/MariaDB is available. Install and activate additional PHP modules to choose other database types.
**For more details check out the documentation. ↗**

nextcloud

•••••••••••••••••••••

nextcloud

localhost

Figure 14.5: Nextcloud configuration page

That's it! Assuming all went well, Nextcloud will set itself up in the background and you'll then continue to the main screen. Since you only created an `admin` account so far, I recommend you create an account for yourself, as well as any friends or colleagues you'd like to check out your Nextcloud server. To do so, go to the top-right corner of the Nextcloud page, where it shows an icon that resembles a gear. When you click on this icon, you'll see an option for **Users**:

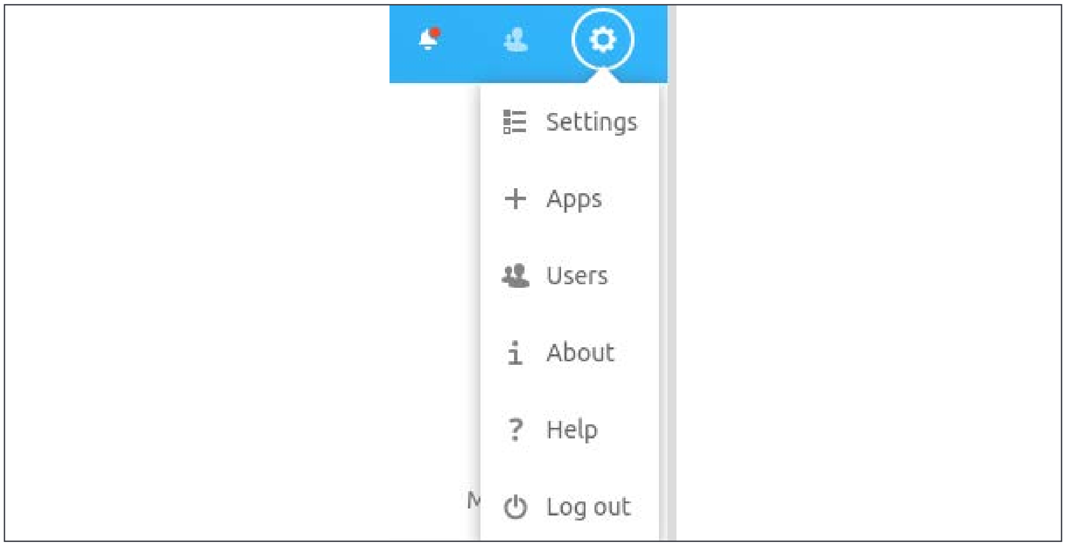
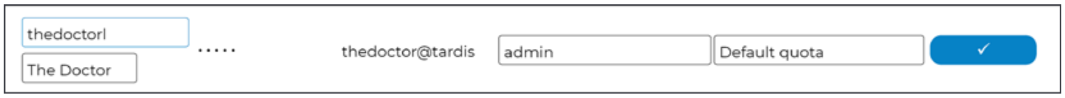


Figure 14.6: Nextcloud menu

On the **Users** screen, you'll be able to add additional users to access Nextcloud. Click on the **New User** button. Simply fill out the **Username** and **Password** fields at the top of the screen, and then click on the blue checkmark icon to finish the process:



thedoctorl ····· thedoctor@tardis admin Default quota ✓

The Doctor

Figure 14.7: Adding a new user to Nextcloud

As an administrative user, you can enable or disable various apps that are used by your users. Out of the box, Nextcloud has a basic suite of apps enabled, such as the **Calendar** and **Mail** plugins. There are many more apps that you can enable in order to extend its functionality. In the top-right corner of the main screen of Nextcloud, you'll find an icon that looks like a gear, and if you click on it, you will find a link to **Apps**, which will allow you to add additional functionality. Feel free to enable additional apps to extend Nextcloud's capabilities. Some of my must-haves include **Notes**, **Reader**, and **Tasks**.

Now, you have your very own Nextcloud server. I find Nextcloud to be a very useful platform. Some Linux desktop environments (such as GNOME) allow you to add your Nextcloud account right to your desktop, which will allow calendar and contact syncing with your computer. I'm sure you'll agree that Nextcloud is a very useful asset to have available. For more information on using Nextcloud, check out the manual. In fact, it's available in the **Files** app from within the application itself.

## Summary

In this action-packed chapter, we looked at serving web pages with Apache. We started out by installing and configuring Apache, and then added additional modules. We also covered the concept of virtual hosts, which allow us to serve multiple websites on a single server, even if we only have a single network interface. Then, we walked through securing our Apache server with TLS. With Apache, we can use self-signed certificates, or we can purchase TLS certificates from a vendor for a fee. We looked at both possibilities. We even set up NGINX, which is a very powerful application that is growing in popularity.

**keepalived** is a handy daemon that we can use to make a service highly available. It allows us to declare a floating IP, which we can use to make an application such as Apache highly available. Should something go wrong, the floating IP will move to another server and as long as we direct traffic toward the floating IP, our service will still be available should the primary server run into an issue. Finally, we closed out the chapter with a guide to installing Nextcloud, which is an application I'm sure you'll find incredibly useful.

In the next chapter of our journey, we'll take a look at the process of automating server configuration with Ansible, which is a lot of fun.

## Further reading

- NGINX documentation: `https://nginx.org/en/docs/`
- NGINX documentation from the Ubuntu community wiki: `https://help.ubuntu.com/community/Nginx`
- Apache HTTP server documentation: `https://nginx.org/en/docs/`
- Nextcloud 19 administration manual: `https://docs.nextcloud.com/server/19/admin_manual/`
- keepalived user guide: `https://keepalived.readthedocs.io/en/latest/`
- How to Setup Let's Encrypt (Certbot) on Ubuntu 20.04: `https://tecadmin.net/how-to-setup-lets-encrypt-on-ubuntu-20-04/`

# 15
# Automating Server Configuration with Ansible

Nowadays, it's not uncommon to have hundreds of servers that make up your organization's infrastructure. As our user base grows, we're able to scale our environment to meet the demands of our customers. As we scale our resources and add additional servers, the amount of time we spend configuring them and setting them up increases considerably. The time spent setting up new servers can be a major burden—especially if we need to create hundreds of servers within a small window of time. As workload demands increase, we need to have a solution in place to manage our infrastructure and quickly deploy new resources with as small a workload as possible. In this chapter, we explore the concept of configuration management along with automated deployments. This sure sounds complicated, but it's not—you'll be surprised how easy it is to automate your configuration.

In this chapter, we will cover:

- Understanding the need for configuration management
- Why Ansible?
- Creating a Git repository
- Getting started with Ansible
- Making your servers do your bidding
- Putting it all together – automating web server deployment
- Using Ansible's pull method

In the introduction, I've already given you some examples of why you might want to consider building automation into your workflow and implement an effective solution in your environment. In the next section, we'll explore the need for automation in more detail before we actually get started.

## Understanding the need for configuration management

When I first started working in the IT industry, it was a much different landscape than it is today. Servers were all physical, and any time you needed a new server, you literally needed to call a vendor and order one. You waited for a week or two for the server to be built and sent to you. When it arrived, you installed it in a rack, set up an operating system, and then installed whatever applications you needed. You then tested the server for a while, to make sure the combination of software, hardware, and drivers was stable and reliable. After some time, you'd deploy the new server into production.

Nowadays, it's still the case that system administrators often need to purchase and install hardware, much like the process I mentioned in the previous paragraph. However, with virtual machines and containers, the physical hardware we install is commonly just a catalyst to host virtual resources. In the past, we had one physical server for each use case, which meant we needed to have very large server rooms. But in modern times, you may have a server with dozens of cores that are capable of running hundreds of virtual machines. But the problem of configuration still remains—the process of setting up an operating system and applications is a very time-consuming endeavor.

As the landscape changed, the need for automation increased. Servers needed to be deployed quickly and efficiently. With the large number of servers in a typical data center, it became less and less practical to connect to each and configure them one by one every time a change was necessary. For example, when a security vulnerability hit the news, the typical administrator would need to manually install a patch on every server. This could take days or even weeks. That's not very efficient.

To better deal with this issue, the concept of configuration management has become very popular. With configuration management, an administrator can write some sort of code (such as a script) and then use a utility to execute it across every server. Configuration management is also known as **Infrastructure as Code** (**IaC**), and basically lets the administrator define a set of guidelines for servers of various types and have them automatically be provisioned to meet those requirements. This automation saves a ton of work.

Configuration management also comes into play while provisioning a new server. Imagine defining some rules for a specific type of server, and having it come to life meeting those exact specifications. The applications you want it to have are installed during the provisioning process, configuration files are copied over, users are created, and firewall rules are put in place, all automatically as defined in your specification. Put even more simply, imagine setting up something like a web server with just a single command. No need to install Apache or do any of that manual work. You simply request a server, and the configuration management solution you have in place will take care of the rest.

IaC, which is basically a fancy term for configuration management, is essentially just the automated running of scripts on your servers. In this book, we've looked at automation already. In *Chapter 6*, *Boosting Your Command-line Efficiency*, we wrote a simple script that we could use to back up a server. That same mentality can be used for provisioning servers as well, by simply having a server run a script when it comes online. For existing servers, you can make a change once and have that change applied to every server you manage, or even just a subset.

This is where configuration management utilities, such as Chef, Puppet, and others, come into play. Each of these solutions features a specific type of scripting language that is designed from the ground up to facilitate the provisioning of resources. With such utilities, there is typically some sort of program (or locally installed agent) that interprets the instructions from a central server and runs them on its clients. Each solution is relatively smart; it will determine what needs to be done and perform the steps. If a requirement is met, the instruction is skipped. If a required resource is not present, it will be configured appropriately. One such configuration management solution is Ansible, which we will use in this chapter.

## Why Ansible?

In this chapter, I will show you how to set up Ansible, and then we will use it to automate some configuration tasks. By the end of this chapter, you'll understand the basic concepts you can use to start the process of automating deployments in your organization. You may be wondering, then, why Ansible and not one of the other solutions, such as Chef or Puppet?

Some configuration management solutions are relatively heavy from a resource perspective. With other solutions, you'll generally have a central server, which will run a master program. This program will periodically *check in* with each server under its control by communicating with the agent installed on each server. Then, the agent will receive instructions from the central server and carry them out.

This means that you'll need to maintain a server with modest CPU and RAM requirements, and the agent on the client side of the communication will also spend valuable CPU in order to carry out the instructions. This resource utilization can be very heavy on both the primary and client servers.

Ansible is very different than the other solutions in that there is no agent at all. There is typically a server, but it's not required to run any resource-intensive software. The entire configuration process happens via SSH, so you can even carry out the instructions from your workstation if you want to skip having to maintain a central server. Typically, the administrator will create a user account on each server, and then the central Ansible server (or workstation) will execute commands over SSH to update the configuration on each machine. Since there is no agent installed on each server, the process takes a lot less CPU. Of course, the instructions that Ansible gives your servers will definitely result in CPU usage, but certainly a lot less than the other solutions.

Ansible is typically set up by creating an *inventory file*, which contains a list of hostnames or IP addresses that Ansible will be instructed to connect to and configure. If you want to add a new server, you simply make sure that a specific user account exists on that server, then you add it to the inventory list. If you want to remove it, you delete the line in the inventory file corresponding to that server. It's very easy.

However, something that's magical about Ansible is that you don't even have to run a central server at all if you don't want to. You can store your Ansible configuration in a Git repository, and have each server download code from the repository and run it locally. This means that if you do have a dynamic environment where servers come and go all the time (which is very common in cloud deployments), you don't have to worry about maintaining an inventory file. Just instruct each server to download the code and provision themselves. This is known as the *pull* method of Ansible, which I will also show you.

While solutions such as Chef and Puppet have their merits and are definitely fun to use, I think you'll find that Ansible scales better and gives you far more control over how these hosts are configured. While it's up to you to figure out exactly how you want to implement Ansible, the creative freedom it gives you is second to none. I've been using Ansible for several years, and I'm still finding new ways to use it. It's a technology that will grow with you.

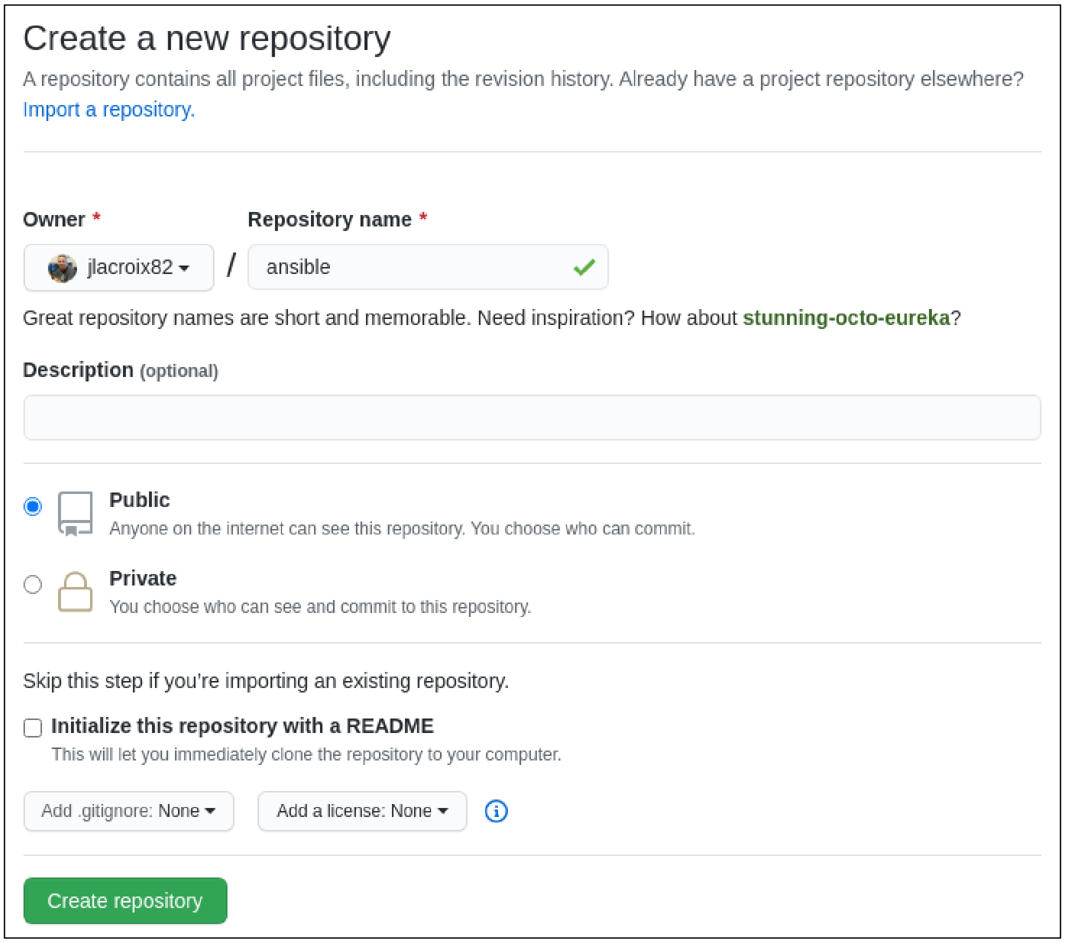
## Creating a Git repository

For the examples in this chapter, it's recommended that you create a Git repository to store your Ansible code. This isn't required, as you can find other ways of hosting your code, but it's highly recommended. This is especially true when we get to the pull method of Ansible at the end of this chapter. In this section, I'll walk you through creating a repository. If you already know how to use GitHub, you can skip this section.

While a full walkthrough of Git is beyond the scope of this book, the basics are more than enough for following along here. When it comes to Git, you can simply install the `git` package on a server to have it host your code, but GitHub is probably the easiest way to get started. An added bonus is that GitHub is home to a lot of great projects you can benefit from, and browsing the code of these projects is a great way to become more accustomed to syntax rules with different scripting and programming languages. For our purposes, we're going to use GitHub as a central place to store our Ansible code.

It probably goes without saying, but GitHub is a public resource. Any code you upload to the service will be available for all to see by default. Therefore, be mindful of the information you commit to your repository. Make sure you don't include any personally identifiable information, passwords, or anything else you don't want the public to know about you or your organization. You can create a private repository to hide confidential information, but it's still safer to not upload protected information at all.

To get started, create an account at `https://www.github.com` if you don't already have one, which is free. Make sure you create a reasonably secure password here. After you create an account, click on **New repository**, and then give it a name (simply calling it `ansible` is fine):



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * jlacroix82 / Repository name * ansible ✓

Great repository names are short and memorable. Need inspiration? How about **stunning-octo-eureka**?

**Description** (optional)

● **Public**
Anyone on the internet can see this repository. You choose who can commit.

○ **Private**
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Figure 15.1: Creating an Ansible repository on GitHub

In the example screenshot, I created a repository that is **Public**, which means exactly that—anyone will be able to view the code. You can create **Private** repositories as well, if you prefer. Since we're not going to include confidential information in the repository during the examples in this book, we don't need to worry about that right now.

Once the repository is created using **Create repository**, we'll need to download it locally. For that, we'll need to install the `git` package:

```
sudo apt install git
```

Next, we should set up our local Git client so that we can fill out our name and email address, otherwise Git will most likely complain. To do this, we can issue the following commands, substituting the content in the quotes with your information:

```
git config user.email "you@example.com"
git config user.name "John Doe"
```

To download our repository, the following will do the trick (ignore the warning about the repository being empty if you see such a message):

```
git clone https://github.com/myusername/ansible.git
```

Now you have the Git repository downloaded locally. Right now, the repository doesn't include anything useful. To create a file within the repository, you simply change your working directory to be inside the repository folder that was downloaded when you cloned it, and create whatever files you want inside. By default, Git doesn't care about any files you create inside the repository until you add them. For example, we can create a test file and commit it to the repository with the following commands:

```
echo "this is a test" > testfile.txt
git add testfile.txt
git commit -m "initial commit"
```

With these commands, we used `echo` to create a test file with some text in it. Then, we used the `git add` command to tell Git that we want the file to be a part of our repository. Finally, we finalized our changes by using `git commit`, along with the `-m` flag and a message about the commit. At this point, the changes only exist locally. To push our changes back to GitHub, we use the following command from inside the repository directory:

```
git push origin master
```

By following the on-screen prompts (GitHub username and password), our changes will be placed inside our actual repository.

So, how does this help us when it comes to configuration management? Typically, the code that administrators use to provision servers will be placed inside a repository for safekeeping. If the local copy of the code is lost, it can simply be cloned again from the repository. GitHub is a safe place to put our code, since we can be reasonably sure that our code won't disappear as the service is very stable (you still may want to create a local backup to be safe). Whether you're using Ansible, Chef, Puppet, or another utility, it's common practice to keep the code in a Git repository. In regards to Ansible, this will directly impact us in the last section of this chapter since we'll be using the `ansible-pull` command, which actually expects a repository URL.

In practice, when we create Ansible playbooks, you should commit those changes back to the repository. I won't specifically call on you to do that, so go ahead and keep that in mind as we go. When you create a new playbook, add it to the repository, then commit it. If you make changes to existing files, commit those changes. Be sure to use the `git push` command to push your changes back to the repository. For example, if you created a file inside the repository named `myplaybook.yml`, you would execute commands such as these:

```
git add myplaybook.yml
git commit -m "insert message about the commit here"
git push origin master
```

Go ahead and practice this a bit before you move on. Even if you don't use Ansible in production, understanding the basics of Git is invaluable, as you'll almost definitely need it at some point in your career.

## Getting started with Ansible

The first thing to know about Ansible is that it changes constantly. New versions with exciting features are released regularly, and it shows no sign of slowing down whatsoever. There is a lot of excitement around this technology, so it's regularly improving. The reason I'm bringing this up is that many distributions offer an older version of Ansible in their repositories, with Ubuntu being no exception. That means that if you simply run `apt install ansible` to get the software from Ubuntu's repositories, you may get an older version, and that version may not work with example solutions you find online. If you run into this situation, you may need to install a newer version of Ansible directly from the vendor's website to overwrite the older version that ships with Ubuntu 20.04.

The examples in this book were created with Ansible 2.9.x in mind, which is the version that ships with Ubuntu 20.04. However, depending on when you are reading this, a new version is most likely available. This shouldn't be an issue with regard to the examples in this chapter, because Ansible handles backward compatibility pretty well. Even a newer version should run the code in this book just fine. However, be on the lookout for examples online that utilize features that may not exist in your version (which is where you may run into a problem). You can install a newer version of Ansible from the official website, but for the purposes of this book, we'll use the version that's in the default repositories.

Go ahead and install `ansible` via `apt`:

```
sudo apt install ansible
```

Now you should have the `ansible-playbook` command available, which is the main binary that is used with Ansible. There are other commands that Ansible provides us, but we're not concerned with those.

In order to follow along with the remainder of this chapter, it's recommended that you have at least two servers to work with; the more the better. If you have a **Virtual Machine** (**VM**) solution such as VirtualBox available, simply create additional VMs. To save time, consider cloning an existing VM a few times (just make sure you don't overload your computer/server by over-allocating resources).

The most common workflow of Ansible works something like this: you have a main server or workstation, on which Ansible is installed. While you don't need an agent on the clients, they will, however, need OpenSSH installed and configured as that's how Ansible communicates. To make things easy, it's recommended to have a dedicated Ansible user on each machine, and the Ansible user on the server should be able to connect to each machine without a password. It doesn't matter what you call the Ansible user; you can simply use `ansible` or something clever. We already covered how to create SSH keys in *Chapter 10*, *Connecting to Networks*, so refer to that if you need a reminder. Creating users was covered in *Chapter 2*, *Managing Users and Permissions*. In a nutshell, here are the things you should work on in order to set up your environment for Ansible:

1. Install Ansible on a central server or workstation
2. Create an Ansible user on each machine you want to manage configuration on
3. Create the same user on your central server or local machine

4. Set up the Ansible user on the server so that it can connect to clients via SSH without a password
5. Configure `sudo` on the client machines so that the Ansible user can execute commands with `sudo` with no password

In previous chapters, we covered how to create users and SSH keys, but we have yet to cover the last point. Assuming you named your Ansible user `ansible`, create the following file:

```
/etc/sudoers.d/ansible
```

Inside that file, place the following text:

```
ansible ALL=(ALL) NOPASSWD: ALL
```

Next, we need to ensure that the file is owned by `root`:

```
sudo chown root:root /etc/sudoers.d/ansible
```

Finally, we need to adjust the permissions of the file:

```
sudo chmod 440 /etc/sudoers.d/ansible
```

Go ahead and test this out. On the server, switch to the `ansible` user:

```
sudo su - ansible
```

Then, to test this out, use SSH to execute a command on a remote machine:

```
ssh 192.168.1.123 sudo ls /etc
```

How does this work? You may or may not know this, but if you use SSH to execute just one command, you don't necessarily need to set up a persistent connection. In this example, we first switch to the `ansible` user. Then, we connect to `192.168.1.123` (or whatever the IP address of the client is) and tell it to execute `sudo ls /etc`. Executing an `ls` command with `sudo` may seem like a silly thing to do, but it's great—it allows you to test whether or not `sudo` works without doing anything potentially dangerous. Listing the contents of a directory is about as innocent as you can get.

It may seem like an awful lot of steps in order to get configuration management working. But make sure you think with a system administrator's mindset—these setup tips can be automated. In my case, I have a Bash script that I run on each of my servers that sets up the required user, keys, and `sudo` access specifically for Ansible. Anytime I want to add a new server to Ansible, I simply run that script on the machine just once, and from that point forward Ansible will take care of the rest.

What should have happened is that the command should have executed and printed the contents of `/etc` without prompting you for a password. If this doesn't work, make sure you completed each of the recommended steps. You should have an `ansible` user on each machine, and that user should have access to `sudo` without a password since we created a file for that user in `/etc/sudoers.d`. If the SSH portion fails, check the log file at `/var/log/auth.log` for clues, as that is where related errors will be saved. Once you have met these requirements, we can get automating with Ansible!

## Making your servers do your bidding

As server administrators, we're control freaks. There are few things more exciting than executing a command and having every single server obey it and carry it out. Now that we have Ansible set up, that's exactly what we're going to do. I'm assuming by now you have some machines you want to configure, and they're all set up to communicate via SSH with your central server. Also, as I mentioned before, I highly recommend you utilize something like Git to store your configuration files, but that's not required for this section.

## Setting up an inventory file and configuring Ansible settings

First, we'll need an inventory file, which is a special text file Ansible expects to find that tells it where to find servers to connect to. By default, the name of this file is simply `hosts` and Ansible expects to find this file in the `/etc/ansible` directory.

There's actually a way to avoid needing to create an inventory file, which we'll get into later in this chapter.

A sample inventory file will be created by default when you install the `ansible` package. If for some reason it doesn't exist, you can create the file easily with the following command:

```
sudo touch /etc/ansible/hosts
```

We should also make sure that only the Ansible user account can read it. Execute the following command to change ownership (replace `ansible` with whatever user account you chose as your Ansible account if you're using something different):

```
chown ansible /etc/ansible/hosts
```

Next, modify the permissions such that only the owner can view or change the file:

```
chmod 600 /etc/ansible/hosts
```

Next, empty the `hosts` file so we start from scratch, and populate the file with the IP addresses of the servers you wish to manage. Feel free to make a backup of the original host file, if it already exists. It should look similar to the following:

```
192.168.1.145
192.168.1.125
192.168.1.166
```

That's it; it's simply just a list of IP addresses. I bet you were expecting some long configuration with all kinds of syntax requirements? Sorry to disappoint. All you need to do is copy a list of IP addresses of the servers you want to manage into this file. If you have DNS names set up for the machines you want to configure, you can use them instead:

```
myhost1.mydomain.com
myhost2.mydomain.com
myhost3.mydomain.com
```

Since Ansible understands IP addresses as well as DNS names, we can use either or a combination of both in order to set up our inventory file. We can also split up our hosts within the inventory file between different roles, but that is outside the scope of this book. I do recommend learning about roles in Ansible if you wish to take your knowledge further (see the *Further reading* section for more information).

If you decide not to store your inventory file at `/etc/ansible/hosts`, you must tell Ansible where to find it. There is another important file to Ansible, and that is its configuration file, located at `/etc/ansible/ansible.cfg`. Inside this file, we can fine-tune Ansible to get the best performance possible. While we won't go over this file in detail, just know that you can seriously increase the performance of Ansible by fine-tuning its settings, and Ansible will read settings from its configuration file every time it runs. In our case, if we wish to store our inventory file somewhere other than `/etc/ansible/hosts`, we will need to add the following two lines to this file (create the file if it doesn't exist already):

```
[defaults]
inventory = /path/to/hosts
```

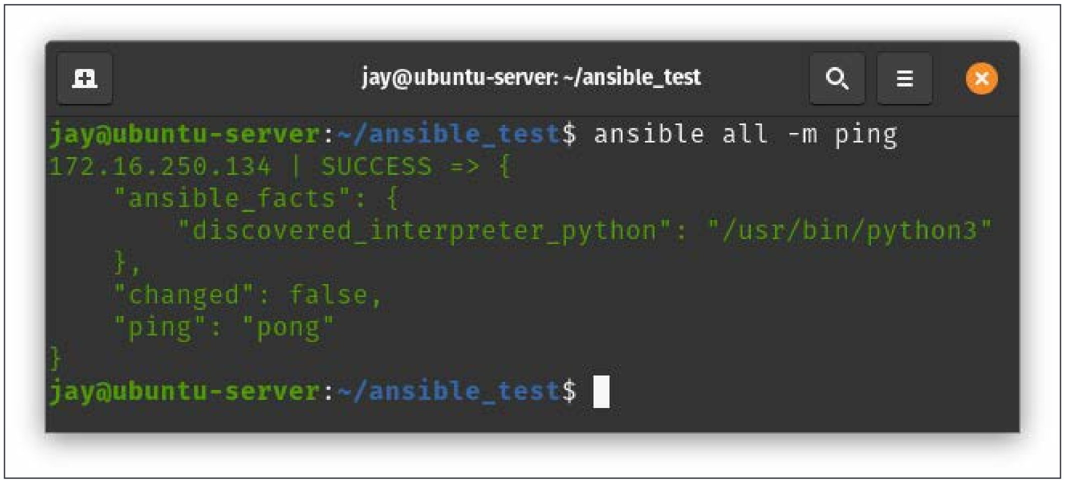
As you can see, we're basically telling Ansible where to find its inventory file. There are many more configuration items we can place in the `ansible.cfg` file to configure it further, but that's all we need to configure right now.

Similar to the inventory file, Ansible also checks the local directory for a file named `ansible.cfg` to fetch its configuration, so you could actually include the configuration file in the Git repository as well, and then execute Ansible commands from within the repository directory. This works because Ansible will check for the existence of a configuration file in your current working directory, and use it if it's found there. You may want to be careful about including your configuration file in your Git directory, though. While it's not as private as the inventory file, it can potentially contain privileged information. Therefore, you may want to keep the file at `/etc/ansible/ansible.cfg` and manage it outside of the Git repository if you include anything private in the file (for example, encryption keys).

Now we can test out whether or not Ansible is working at this point. Thankfully, this is also easy. Just simply execute the following command:

```
ansible all -m ping
```

The results should look similar to this:



```
jay@ubuntu-server:~/ansible_test$ ansible all -m ping
172.16.250.134 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
jay@ubuntu-server:~/ansible_test$
```

Figure 15.2: Testing Ansible

Depending on how many servers you have set up, you should see that output one or more times. You may think that all that command has done is a simple ping test, but `ping` means something different to Ansible than usual. It's actually attempting to make a connection to the server, to test its availability. If it fails, double-check that the hosts are available over SSH. Success here simply means that Ansible is able to communicate with your hosts via SSH. Now that the communication exists, we can build some actual configuration.

## Configuring client servers

Ansible uses something called a **playbook** in order to store configuration. A playbook is essentially another name for a file in the YAML format. YAML is beyond the scope of this book, but you don't have to master this format or even fully understand it to use it with Ansible. That will automatically come in time. The takeaway here is that YAML is simply the format that Ansible uses. A playbook is basically just a collection of instructions written in this format, and each individual instruction is known as a play. You can think of this with the analogy of a sport, like football. Although I don't know the first thing about football, I do know that football coaches have playbooks containing things that they want their players to do, and each action by a player is a play. It's the same concept here.

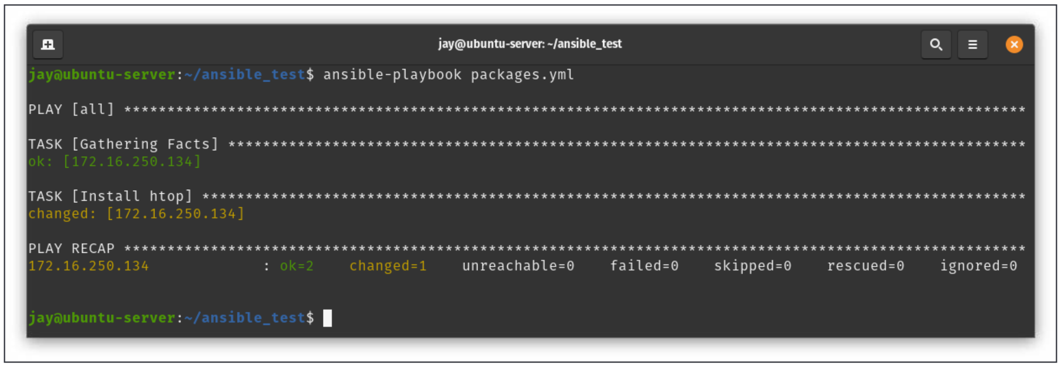
Let's write our first playbook. Create a file called `packages.yml` in your local Ansible directory. You can fill it with this content (make sure you include the hyphens):

```
---

- hosts: all
  become: true

  tasks:
  - name: Install htop
    apt:
      name: htop
```

We can run this playbook with the following command:

```
ansible-playbook packages.yml
```

This will produce an output that looks something like the following:



```
jay@ubuntu-server:~/ansible_test$ ansible-playbook packages.yml

PLAY [all] ****************************************************************************************************

TASK [Gathering Facts] ****************************************************************************************
ok: [172.16.250.134]

TASK [Install htop] *******************************************************************************************
changed: [172.16.250.134]

PLAY RECAP ****************************************************************************************************
172.16.250.134             : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

jay@ubuntu-server:~/ansible_test$
```

Figure 15.3: Ansible in action

Just like that, all of the hosts in your inventory file will have the `htop` package installed. It really doesn't matter which package you install, so long as it exists in the repositories; I just used `htop` as a simple example. But once you run it, you should see an overview as far as what was changed. Ansible will tell you how many items your hosts updated, how many tasks have failed, and how many targets weren't reachable at the time the playbook was run.

Let's take a closer look at what the instructions in this sample playbook do. The hyphens at the beginning are part of the YAML format, so we really don't need to get into that. Spacing is very important with the YAML format, as you need to be consistent throughout. In my examples, I am inserting two spaces underneath each heading. A heading starts with a hyphen:

```
- hosts: all
```

Here, we declare which hosts we want to have the commands apply to. I added `all` here, which basically runs the configuration against every host in the inventory file. With advanced usage, you can actually create `roles` in Ansible and divide your hosts between them, such as a web server, database server, and so on. Then, you can apply configuration only to hosts inside a particular role. We're not going to get into that in this chapter, but just know that it is possible.

The next line is `become`:

```
become: true
```

This line is basically Ansible's term for describing `sudo`. We're telling Ansible to use `sudo` to execute the commands, since installing packages requires `root` privileges. The next part of the playbook is as follows:

```
tasks:
```

This line starts the next section, which is where we place our individual tasks. Next, we name our new play:

```
- name: Install htop
```

With `name`, we give the play a name. This isn't required but you should always include it. The importance of this is that whatever we type here is what is going to show up in the logs if we enable logging, and will also print to the terminal as the play runs. We should be descriptive here, as it will certainly help if a play fails and we need to locate it in a log file that has hundreds of lines. Next, we utilize the `apt` module and tell it to install a package, `htop` in this case:

```
apt:
  name: htop
```

We use the `apt` module simply because Ubuntu uses the `apt` command to manage packages, but modules exist for all of the popular Linux distributions. Ansible's support of package managers among various distributions is actually fairly extensive.

All of the major distributions, such as Red Hat, Fedora, openSUSE, Arch Linux, Debian, and others are supported (and those are just the ones I've used in my lab off the top of my head). If you want to execute a play against a server that's running a distribution other than Ubuntu, simply adjust `apt` to something else, such as `yum` or `dnf`.

You can, of course, add additional packages by simply adding more plays to the existing playbook:

```
---

- hosts: all
  become: true

  tasks:
  - name: Install htop
    apt:
      name: htop

  - name: Install git
    apt:
      name: git

  - name: Install vim-nox
    apt:
      name: vim-nox
```

However, that's not an extremely efficient method. I'll show you how we can combine multiple similar plays in one play. Sure, you don't have to, but I think you'll agree that this method looks cleaner:

```
---

- hosts: all
  become: true

  tasks:
  - name: Install packages
    apt:
      name:
        - git
        - htop
        - vim-nox
```

With the new format, we include just one play to install multiple packages. This is similar to the concept of a *for loop* if you have programming knowledge. For every package we list, it will run the `apt` module against it. If we want to add additional packages, we just add a new one to the list. Simple.

We can also copy files to our hosts. Consider the following example playbook, which I will call `copy_files.yml`:

```
---

- hosts: all
  become: true
  tasks:

  - name: copy SSH motd
    copy:
      src: motd
      dest: /etc/motd
```

Then, you can run this playbook with the following command:

```
ansible-playbook copy_files.yml
```

Inside the same directory, create a file called `motd` and place any text in it. It doesn't really matter what you type into the file, but this file in particular acts as a message that is printed any time a user logs into a server. When you run the playbook, it will copy that file over to the server at the destination you configured. Since we created a message of the day (`motd`), we should see the new message the next time we log in to the server.

By now, you're probably seeing just how useful Ansible can be. Sure, we only installed a few packages and copied one file. We could've performed those tasks easily ourselves without Ansible, but this is only a start. Ansible lets you automate everything, and we have to start somewhere. You can tell it to not only install packages and copy files, but you can also use it to start services, apply configuration file templates, and much more—it will surprise you. In fact, you can go as far as to automate the setup of a web server, a user's workstation... you name it!

## Putting it all together – automating web server deployment

Speaking of automating the setup of a web server, why don't we go ahead and do exactly that? It'll be another simple example, but it will serve you well if we demonstrate more of what Ansible can do. We will set up a playbook to perform the following tasks:

1. Install Apache
2. Start the `apache2` service
3. Copy an HTML file for the new site

First, let's set up the playbook to simply install Apache. I called mine `apache.yml`, but the name is arbitrary:

```
---
- hosts: all
  become: true
  tasks:

  - name: Install Apache
    apt:
      name: apache2
```

No surprises here, we've already installed a package at this point. Let's add an additional instruction to start the `apache2` service:

```
---

- hosts: all
  become: true
  tasks:

  - name: Install Apache
    apt:
      name: apache2

  - name: Start the apache2 services
    service:
      name: apache2
      state: started
```

So far, the syntax should be self-explanatory. Ansible has a `service` module, which can be used to start a service on a host. In this case, we start the `apache2` servicee (the service would've already been started when `apache2` was installed, but at least this way we can make sure it's started). You do have to know the service name ahead of time, but you don't have to pay attention to what utility needs to be used in the background to start the service. Ansible already knows how to start services on all the popular distributions and takes care of the specifics for you in the background.

Having a play that starts `apache2` may seem a bit redundant since most packages you install on an Ubuntu server will automatically start the associated service as soon as it's installed. But when we write automation code, it's important to be clear and explicit about what the desired end result is supposed to be. Even though `apache2` will automatically start once Ansible installs the package, we're adding a `service` play to clarify the fact that it needs to be running, so anyone looking at it will know what's expected. Also, we can make sure `apache2` is `enabled`, by adding one more line to that section of the code (note the bold line):

```
- name: Start the apache2 services
    service:
    name: apache2
    state: started
    enabled: true
```

I'll leave it up to you whether or not to include that extra line, but the takeaway is to be as clear and direct as you can when writing automation, so there's no confusion about what the end result is supposed to be.

Well, that was easy. Let's create a simple web page for Apache to serve for us. It doesn't need to be fancy, we just want to see that it works. Create the following content inside a file named `index.html` in the same working directory as the other Ansible files:

```
<html>
<title>Ansible is awesome!</title>
<body>
    <p>Ansible is amazing. With just a small text file, we automated
the setup of a web server!</p>
</body>
</html>
```

As you can see, that HTML file is fairly lame, but it will work just fine for what we need. Next, let's add another instruction to our Apache playbook:

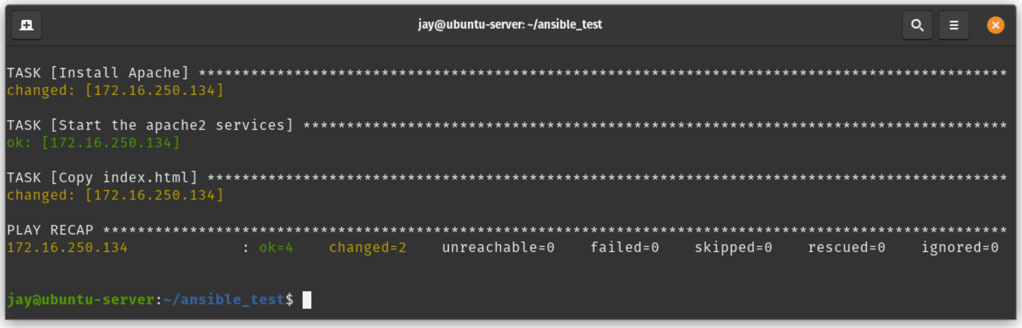
```
---

- hosts: all
  become: true
  tasks:

  - name: Install Apache
    apt:
     name: apache2

  - name: Start the apache2 services
    service:
      name: apache2
      state: started

  - name: Copy index.html
    copy:
      src: index.html
      dest: /var/www/html/index.html
```

With the `copy` module, we can copy a file from our local Ansible directory to the server. All we need to do is provide it with a source (`src`) and destination (`dest`).

Let's go ahead and run the new playbook:

```
ansible-playbook apache.yml
```

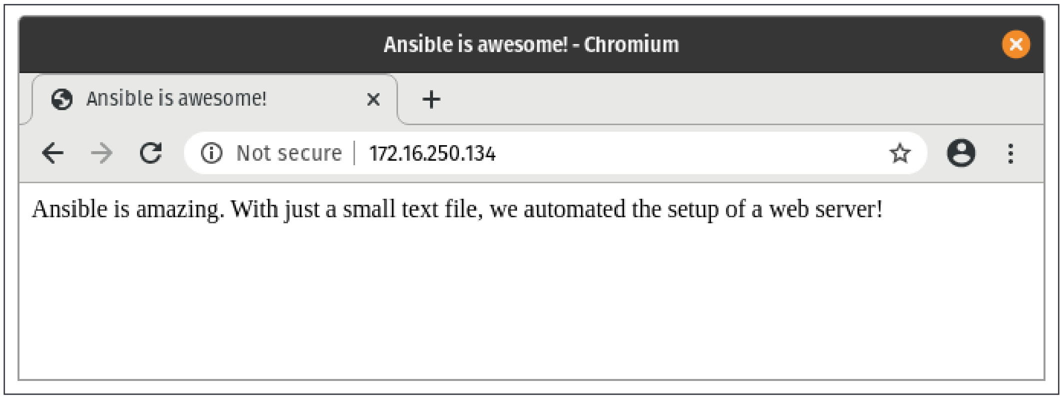
This will produce an output like the following:



```
TASK [Install Apache] *********************************************************************************
changed: [172.16.250.134]

TASK [Start the apache2 services] *********************************************************************
ok: [172.16.250.134]

TASK [Copy index.html] ********************************************************************************
changed: [172.16.250.134]

PLAY RECAP ********************************************************************************************
172.16.250.134             : ok=4    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

jay@ubuntu-server:~/ansible_test$ 
```

Figure 15.4: Installing Apache and copying a default site file with Ansible

Within a few minutes, you should have at least one web server configured by Ansible. In a real production environment, you would've only run this on servers within a specific role, but roles are beyond the scope of this chapter. But from the simple playbook we created, you should be able to see the power of this amazing piece of software:



Ansible is amazing. With just a small text file, we automated the setup of a web server!

Figure 15.5: An example of a web page provisioned by Ansible

At this point in our exploration into configuration management with Ansible, we've installed packages, started services, and copied files. Admittedly, this isn't much, but it's really all you need in order to practice the basics. The documentation for Ansible includes guides on how to do all sorts of things, and you'll be able to utilize the various modules it provides to perform many different tasks.

To explore Ansible further, I recommend that you think about things you do on a regular basis that you would benefit from automating. Things like installing security updates, creating user accounts, setting passwords, and enabling services to automatically start at boot time are tasks that are easy to get started with.

In addition, you may want to consider adding a simple **cron job** to run under your `ansible` user, to run the playbook every hour or so. We covered cron jobs in *Chapter 7, Controlling and Managing Processes*. Adding a cron job shouldn't add any overhead from a resource perspective, since Ansible won't actually be doing much unless you add a new command. In more advanced usage, you'll want to have Ansible check out code from a repository and then apply the configuration if it has changed. This way, all you have to do is commit to a Git repository and all of your servers will download the configuration and run it at the next scheduled time. One of the things I love most about Ansible is that it's easy to get started, but you'll continue to find new ways to use it and benefit from it.

## Using Ansible's pull method

The way we set up our Ansible configuration in the previous section works very well if we have a list of specific servers that we want it to manage. To add a new server, we create the user account and set up the SSH configuration on the new host, and then add it to the inventory file. If we decommission that server, we simply remove it from the inventory file. This works well in a static environment, where servers you deploy typically stay around for a while. In a dynamic environment, though, this may not work as well.

Dynamic environments are very typical in the cloud. With cloud computing, you typically have one or more virtual servers that provide a service to your company or users. These servers may come and go at any time. With dynamic environments, servers will come online as needed to handle load, and will also get decommissioned automatically as load decreases. Therefore, you never really know when a server is going to come online, and having to manually provision a server in such an environment is inefficient.

For this reason, Ansible's inventory file may not be a good fit for dynamic infrastructure. There certainly are ways to make Ansible's inventory work in such an environment, as you can actually replace the inventory file with an executable script that can make API calls and customize itself for your infrastructure if you so desire. However, that's out of the scope of this book, and there's an easier method anyway.

As we've seen, Ansible uses an inventory file, and connects to every server listed in that file. However, Ansible also features a *pull mode*, where instead of having a central server that connects to other machines, each server in pull mode will actually run Ansible against itself. In my opinion, this is a great way to use Ansible and it doesn't seem to get the attention it deserves. First, I'll explain the theory of how it works, and then we can work through an actual example.

With pull mode, you'll want to have your Ansible playbooks inside a Git repository. This repository must be accessible from the servers you will manage. For example, if you store the Git repository on GitHub, you'll want to make sure the servers can access GitHub externally. If you host your own Git server internally, you'll want to make sure your servers are able to access it through your firewall or any security rules you may have in place.

Pull mode is used with the `ansible-pull` command, which actually comes bundled with Ansible. The syntax looks like the following:

```
ansible-pull -U https://github.com/myusername/ansible.git
```

Of course, you'd replace the URL with the actual HTTP or HTTPS URL to your Git repository. However, that's basically it. The `ansible-pull` command simply expects the `-U` option (short for URL) along with the URL to a Git repository.

In order for this to work, you'll need a playbook inside the repository with a special name, `local.yml`. If you don't declare a specific playbook with Ansible, it will expect to find a playbook with that name inside the root of the repository. If you choose to use a name for the main playbook as something other than `local.yml`, you'll need to specify it:

```
ansible-pull -U https://github.com/myusername/ansible.git myplaybook.
yml
```

In this example, the `ansible-pull` command will cache the Git repository located at the specified URL locally, and run the playbook `myplaybook.yml` that you would have inside the repository. One thing you may find is that Ansible might complain about not finding an inventory file, even though that's the entire point of the `ansible-pull` command. You can ignore this error. This will likely be fixed in Ansible at some point in the future, but as of the time of this writing, it will print a warning if it doesn't detect an inventory file.

With the theory out of the way, let's work through an actual example. If you've been following along so far, we created a playbook in the previous section that automates the deployment of a hypothetical web server. We can reuse that code. However, it's best practice to have a file with the name of `local.yml`, so you can simply rename the `apache.yml` playbook we created earlier to `local.yml`. There's one small change we need to make to the file, which I've highlighted here:

```
---

- hosts: localhost
  become: true
  tasks:

  - name: Install Apache
    apt: name=apache2

  - name: Start the apache2 services
    service:
      name: apache2
      state: started
  - name: Copy index.html
    copy:
      src: index.html
      dest: /var/www/html/index.html
```

Since we're executing the playbook locally (without SSH) we changed the `hosts`: line to point to `localhost`, to instruct Ansible that we want to execute the commands locally rather than remotely. Now, we can push this playbook to our Git repository and execute it directly from the repository URL.

Pay careful attention to the `hosts:` line of any playbook you intend to run. If you are using the pull method, this line will need to be changed from `hosts: all` to `hosts: localhost`, the reason being, we are executing the playbooks directly on localhost, rather than from a remote SSH connection. If you don't make this change, you'll see an error similar to the following:

```
ERROR! Specified hosts and/or --limit does not match any hosts
```

Before you run the playbook, you'll want to first switch to your Ansible user, since the playbook will need to be run as a user with `sudo` privileges since it will execute system-level commands:

```
sudo su - ansible
```

Then, execute the playbook:

```
ansible-pull -U https://github.com/myusername/ansible.git
```

If we kept the name as `apache.yml`, we would just specify that:

```
ansible-pull -U https://github.com/myusername/ansible.git apache.yml
```

Keep in mind that since `ansible-pull` executes playbooks directly on localhost, the playbook must be executed by a user that has access to `sudo`. If `sudo` is not configured to run as that user without a password, the playbook will fail as it's not interactive (it won't ask for the password). You can also use `sudo` in front of the `ansible-pull` command and provide the password before it runs, but that won't work if you set it up to run automatically via cron.

If all goes according to plan, the playbook repository should be cached to the server, and the instructions carried out. If there is an error, Ansible is fairly good about presenting logical error messages. As long as the user has permission to execute privileged commands on the server, the required files are present in the repository, and the server has access to download the repository (a firewall isn't blocking it). then the playbook will run properly.

When it comes to implementing the pull method in production across various server types, there are several ways we can go about this. One way is to have a separate playbook per server type. For example, perhaps you'd have an Apache playbook, as well as playbooks specific to database servers, file servers, user workstations, and so on. Then, depending on the type of server you're deploying, you'd specify the appropriate playbook when you called the `ansible-pull` command. If you're using a service such as cloud computing, you can actually provide a script for each server to execute upon creation. You can then instruct the service to automatically run the `ansible-pull` command any time a new server is created. In AWS, for example, you can use a feature known as user data to place a script for a server to execute when it's launched for the first time. This saves you from having to provision anything manually. For this to work, you would first include a command for the server to install Ansible itself, and then the next command would be the `ansible-pull` command along with the URL to the repository. Just those two lines would completely automate the installation of Ansible and the application of your playbook. Try to think of the possibilities ahead of time so you can understand the many ways that automation can benefit you.

While provisioning new servers properly and efficiently is very important, so too is maintaining your existing servers. When you need to install a new package or apply a security update, the last thing you want to do is connect to all of your servers manually and update them one by one. The `ansible-pull` command allows for simple management as well; you just simply run the command again. Every time you run `ansible-pull`, it will download the latest version of the code within your repository and run it. If there are any changes, they will be applied, while things that have already been applied will be skipped. For example, if you include a play to install the `apache2` package, Ansible won't reinstall the package if you run the playbook a second time; it will skip that play since that requirement has already been met.

One trick worth knowing with `ansible-pull` is the `-o` option. This option will ensure that the playbook inside the repository is only run if there have been any actual changes to the repository. If you haven't committed any changes to the repository, it will skip the entire thing. This is very useful if you set up the `ansible-pull` command syntax to be run periodically via cron, for example, every hour.

If you don't include the `-o` option, Ansible will run the entire playbook every hour. This will consume valuable CPU resources for no good reason at all. With the `-o` option, Ansible will only use as much CPU as required for simply checking the repository for changes. The playbook will only be run if you actually commit changes back to the repository.

The introduction to Ansible within this chapter has been very basic, as we've only used the very core of the required components. By looking deeper into Ansible, you'll find more advanced techniques, and some more clever ways to implement it. Examples include automating firewall rules, security patches, and user passwords, as well as having Ansible send you an email any time a server is successfully provisioned (or even when that fails). Basically, just about anything you can do manually, you can automate with Ansible. In the future, I recommend looking at server administration from an automation mindset. As I've mentioned several times before, if you will need to perform a task more than once, automate it. Ansible is one of the best ways of automating server administration.

## Summary

In this chapter, we took a look at configuration management using Ansible. Ansible is an exciting technology that is exploding in popularity. It gives you the full power of configuration management utilities such as Chef or Puppet, without all the resource overhead. It allows you to automate just about everything. During our exploration, we walked through installing packages, copying files, and starting services. Near the end of the chapter, we worked through an example of using Ansible to provision a simple web server, and even explored the pull method, which is very useful in dynamic environments. These concepts form the basis of knowledge that can be expanded to automate more complex rollouts.

The next chapter will be fun: we'll set up our very own virtualization server with KVM. This is one of my favorite topics, and I'm sure you'll enjoy it too. See you there!

## Further reading

- Ansible documentation: `http://docs.ansible.com/ansible/latest/index.html`
- Ansible Roles documentation: `https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html`
- Ansible config file documentation article: `http://docs.ansible.com/ansible/latest/intro_configuration.html`

- `ansible-pull` documentation: `https://docs.ansible.com/ansible/2.4/ansible-pull.html`
- How to manage your workstation configuration with Ansible: `https://opensource.com/article/18/3/manage-workstation-ansible`
- Git basics: `https://git-scm.com/book/en/v2/`
- Setting up Git (GitHub): `https://help.github.com/articles/set-up-git/`
- Getting Started with Ansible (LearnLinuxTV): `https://learnlinux.link/learn-ansible`

# 16
# Virtualization

There have been a great many advancements in the IT space in the last few decades, and a few technologies have come along that have truly revolutionized the technology industry. I'm sure few would argue that the internet is by far the most revolutionary technology to come around, but another technology that has created a paradigm shift in IT is virtualization. This concept changed the way we maintain our data centers, allowing us to segregate workloads into many smaller machines being run from a single server. This allows us to get even more use out of our hardware. Since Ubuntu features the latest advancements of the Linux kernel, virtualization support is actually built right into it. After installing just a few packages to allow us to interact with the virtualization features, we can create virtual machines on our Ubuntu server without the need for a pricey license agreement or support contract. In this chapter, I'll walk you through setting up your own Ubuntu-based virtualization solution. Along the way, I'll walk you through the following topics:

- Prerequisites and considerations
- Setting up a virtual machine server
- Creating virtual machines
- Bridging the virtual machine network
- Simplifying virtual machine creation with cloning
- Managing virtual machines via the command line

In order to get started, we'll need a server to use for this task, and we'll first have some discussion on some considerations to make when setting up a server for this purpose.

## Prerequisites and considerations

I'm sure many of you have already used a virtualization solution before. In fact, I bet a great many readers are following along with this book while using a **Virtual Machine** (**VM**) running in a solution such as VirtualBox, Parallels, VMware, or one of the others. Those applications and others like them are great for testing Ubuntu or other operating systems on your desktop or laptop. In this section, we'll set up a VM server that can act as a centrally available server on which to run VMs.

This will be easier than you may think—Ubuntu has virtualization built right in. This comes in the form of a dynamic duo consisting of **Kernel-based VM** (**KVM**) and **Quick Emulator** (**QEMU**), which together form a virtualization suite that enables Ubuntu (and Linux in general) to run VMs without the need for a third-party solution. KVM is the feature that is built right into the Linux kernel that performs the magic under the hood. It handles the low-level instructions in the kernel that are needed to separate tasks between those run on a physical host and on a guest VM. QEMU is also important, as it emulates hardware components that are generally found in physical servers. The combination of KVM and QEMU make up the virtualization solution that can be enabled on an Ubuntu server to turn it into a host for VMs.

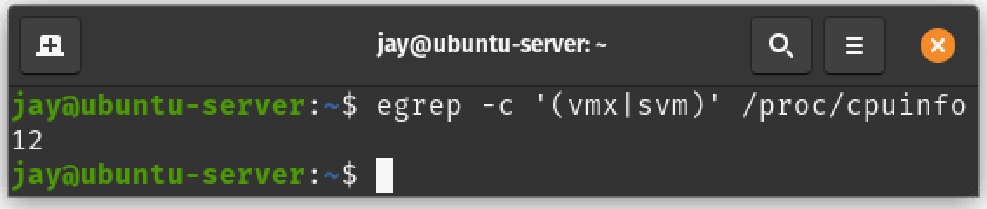
To be fair, you can set up something like VirtualBox on your Ubuntu Server to accomplish the same thing, and end up with a centrally-available virtualization server. And that's perfectly valid, there's certainly nothing wrong with running VirtualBox this way, and many people do. But there are improvements to be had by utilizing a built-in system, and KVM offers a very fast interface to the Linux kernel to run your VMs with near-native speeds, depending on your use case. QEMU/KVM (which I'll refer to simply as KVM going forward) is about as native as you can get.

I bet you're eager to get started, but there are a few quick things to consider before we dive in. First, of all the activities I've walked you through in this book so far, setting up our own virtualization solution will be the most expensive from a hardware perspective. The more VMs you plan on running, the more resources your server will need to have available (especially RAM). Thankfully, most computers nowadays ship with 8 GB of RAM at a minimum, with 16 GB or more being fairly common. With most modern computers, you should be able to run VMs without too much of an impact. Depending on what kind of machine you're using, the CPU and RAM may present bottlenecks, especially when it comes to legacy hardware.

For the purposes of this chapter, it's recommended that you have a PC or server available with a processor that's capable of supporting VM extensions. A good majority of CPUs on computers nowadays offer this, though some may not. To be sure, you can run the following command on the machine you intend to host KVM VMs on in order to find out whether your CPU supports virtualization extensions:

```
egrep -c '(vmx|svm)' /proc/cpuinfo
```

A result of `1` or more means that your CPU does support virtualization extensions. A result of `0` means it does not:



```
jay@ubuntu-server:~$ egrep -c '(vmx|svm)' /proc/cpuinfo
12
jay@ubuntu-server:~$ 
```

Figure 16.1: Checking the CPU for compatibility with virtualization

Even if your CPU does support virtualization extensions, it's often the case that it's disabled by default with most end-user PCs sold today, and even some servers. To enable these extensions, you may need to enter the BIOS setup screen for your computer and enable the option. Depending on your CPU and chipset, this option may be named something similar to "virtualization support," under a more technical name such as VT-x or AMD-V, or other verbiage. Unfortunately, I won't be able to walk you through how to enable the virtualization extensions for your hardware, since the instructions will differ from one machine to another. If in doubt, refer to the documentation for your hardware.

One final note: I'm sure many of you are using VirtualBox, as it seems to be a very popular solution for those testing out Linux distributions (and rightfully so, it's great!). However, you can't run both VirtualBox and KVM VMs on the same machine simultaneously. You can certainly have both solutions installed on the same machine, but you just can't have a VirtualBox VM up and running, and then expect to also be able to start up a KVM VM. The virtualization extensions of your CPU can only work with one solution at a time.

Another consideration to bear in mind is the amount of space the server has available, as VMs can take quite a bit of space. The default directory for KVM VM images is `/var/lib/libvirt/images`. If your `/var` directory is part of the `root` filesystem, you may not have a lot of space to work with here. One trick is that you can mount an external storage volume to this directory, so you can store your VM disk images on another volume. Or, you can simply create a symbolic link that will point this directory somewhere else. We discussed symbolic links in *Chapter 5*, **Managing Files and Directories**. The choice is yours. If your `root` filesystem has at least 10 GB available, you should be able to create at least one VM without needing to configure the storage. I think it's a fair estimate to assume at least 10 GB of hard drive space per VM.

## Setting up a virtual machine server

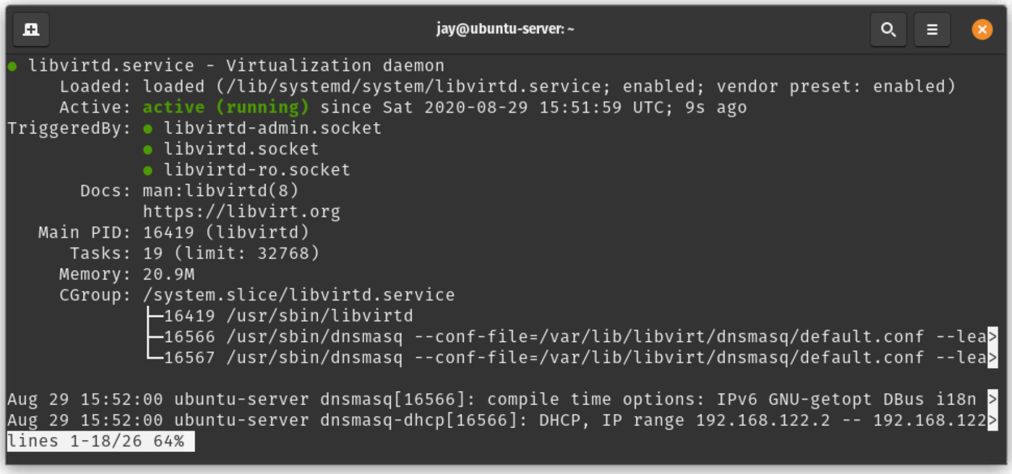
With all the discussion out of the way, let's start the process and set up our virtualization server. Even though KVM is built into the Linux kernel, we'll still need to install some packages in order to properly interface with it. These packages will require a decent number of dependencies, so it may take a few minutes for everything to install:

```
sudo apt install bridge-utils libvirt-clients libvirt-daemon-system
qemu-kvm
```

You'll now have an additional service running on your server, `libvirtd`. Once you've finished installing KVM's packages, this service will be started and enabled for you. Feel free to take a look at it to see for yourself:

```
systemctl status libvirtd
```

You should see information on the state of the service, similar to the following:



```
● libvirtd.service - Virtualization daemon
     Loaded: loaded (/lib/systemd/system/libvirtd.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2020-08-29 15:51:59 UTC; 9s ago
TriggeredBy: ● libvirtd-admin.socket
             ● libvirtd.socket
             ● libvirtd-ro.socket
       Docs: man:libvirtd(8)
             https://libvirt.org
   Main PID: 16419 (libvirtd)
      Tasks: 19 (limit: 32768)
     Memory: 20.9M
     CGroup: /system.slice/libvirtd.service
             ├─16419 /usr/sbin/libvirtd
             ├─16566 /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf --lea>
             └─16567 /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf --lea>

Aug 29 15:52:00 ubuntu-server dnsmasq[16566]: compile time options: IPv6 GNU-getopt DBus i18n >
Aug 29 15:52:00 ubuntu-server dnsmasq-dhcp[16566]: DHCP, IP range 192.168.122.2 -- 192.168.122>
lines 1-18/26 64%
```

Figure 16.2: Checking the status of the libvirtd unit after installing KVM-related packages

Let's stop this service for now, as we have some additional configuration to do:

```
sudo systemctl stop libvirtd
```

Next, we'll need to make sure we have two required groups on our server, `kvm` and `libvirt`. It's quite possible that the packages that we've installed have added these groups on our server already, so feel free to check the contents of `/etc/group` and see if they're there. If not, you can create them with the `groupadd` command:

```
sudo groupadd kvm
sudo groupadd libvirt
```

Our primary user account should be a member of both groups. If your user isn't already a member of these, add your user to the required groups (substitute the username, `jay`, with yours):

```
sudo usermod -aG kvm jay
sudo usermod -aG libvirt jay
```

At this point, you may as well log out and then log in again to ensure the changes to your group memberships have taken effect.

To ensure we'll be able to manage virtualization properly, we should ensure that users of the `kvm` group have access to the `/var/lib/libvirt/images` directory so that they'll have access to the data that will be stored in the directory. First, we'll apply the `kvm` group to this folder:

```
sudo chown :kvm /var/lib/libvirt/images
```

Then, we'll set the permissions of `/var/lib/libvirt/images` such that anyone in the `kvm` group will be able to modify its contents:

```
sudo chmod g+rw /var/lib/libvirt/images
```

With the initial packages and permissions in place, we can now start the `libvirtd` service:

```
sudo systemctl start libvirtd
```

Next, check the status of the service to make sure that there are no errors:

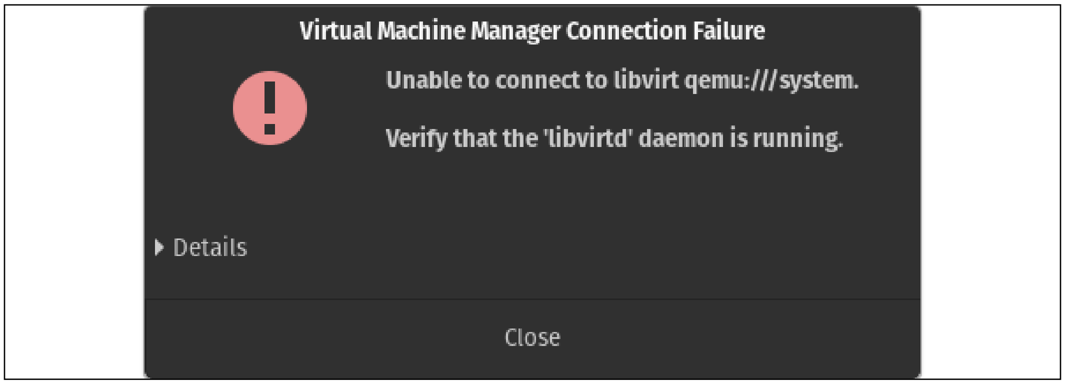
```
sudo systemctl status libvirtd
```

Now that we've configured the server, we can set up our workstation to be able to connect to it and manage the virtualization implementation that we've set up. We'll install a utility that will give us a **graphical user interface** (**GUI**) through which we can perform administration tasks relating to VMs. The utility we'll be using for this purpose is known as Virtual Machine Manager abbreviated as `virt-manager`. This utility is installed on Linux workstations, so you'll need to install it on a laptop or desktop that's running a desktop variant of Linux. If you have a computer running Debian or Ubuntu, the following command will install the packages that are required for this:

```
sudo apt install ssh-askpass virt-manager
```

If you use a distribution of Linux other than Ubuntu or Debian or one based on them, then you may need to consult the documentation for your distribution in order to install `virt-manager`. If you're not running Linux on your workstation at all, there is a suite of command-line utilities that can be used to manage VMs that we'll cover later in this chapter when we discuss this in the *Managing virtual machines via the command line* section. If all else fails, you can install this utility inside a Linux VM running on your workstation.

Next, open `virt-manager` on your administration machine. It should be located in the **Applications** menu of your desktop environment, usually under the **System Tools** section of **Virtual Machine Manager**. If you have trouble finding it, simply run `virt-manager` at your shell prompt. When you first launch it, you may see the following error:


Virtual Machine Manager Connection Failure

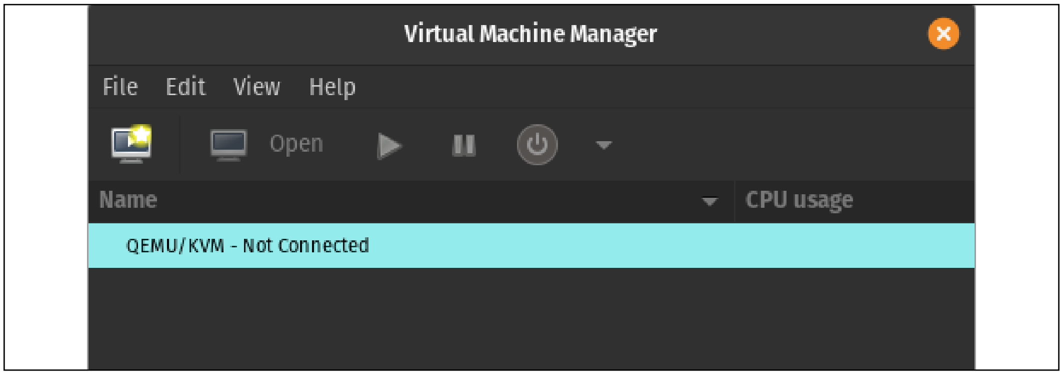
Unable to connect to libvirt qemu:///system.

Verify that the 'libvirtd' daemon is running.

Figure 16.3: A possible error that may appear when first launching virt-manager

If you do see the error, simply dismiss it and don't worry about it. By default, `virt-manager` defaults to attempting to connect to an instance of `libvirtd` running on your local computer. Unless you are also running KVM VMs locally and you've already set it up, this attempt will fail. But that doesn't matter for us, as we'll be using `virt-manager` to manage VMs on our server.

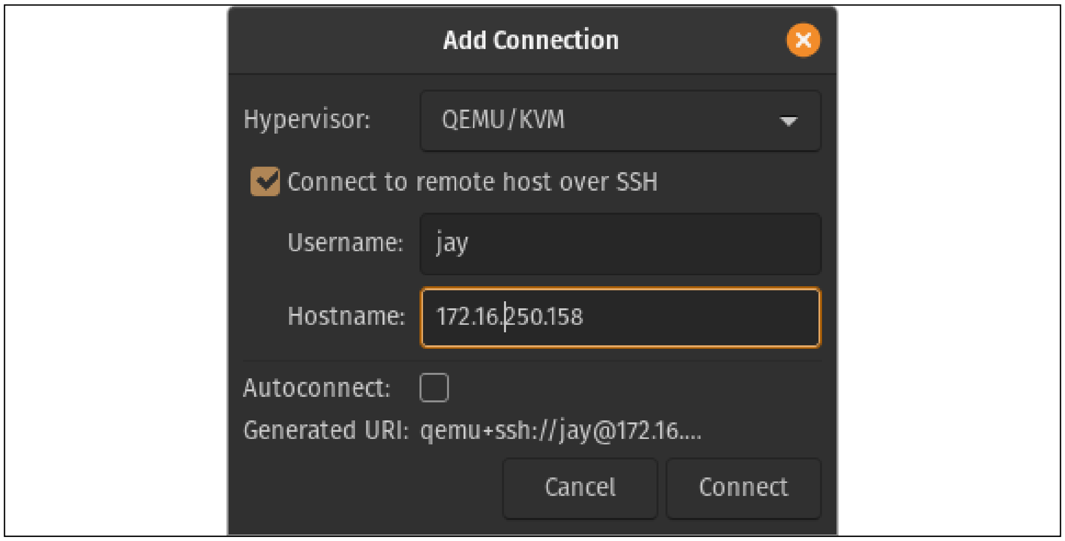
Once you've opened `virt-manager`, you'll see the main window, which will look similar to the following:



QEMU/KVM - Not Connected

Figure 16.4: The virt-manager application

The `virt-manager` utility is especially useful as it allows us to manage both remote and local KVM servers. From one utility, you can create connections to any of your KVM servers, including one or more external servers or `localhost` if you are running KVM on your laptop or desktop. To create a new connection, click on **File** and select **Add Connection**. A new screen will appear, where we can fill out the details for the KVM server we wish to connect to:

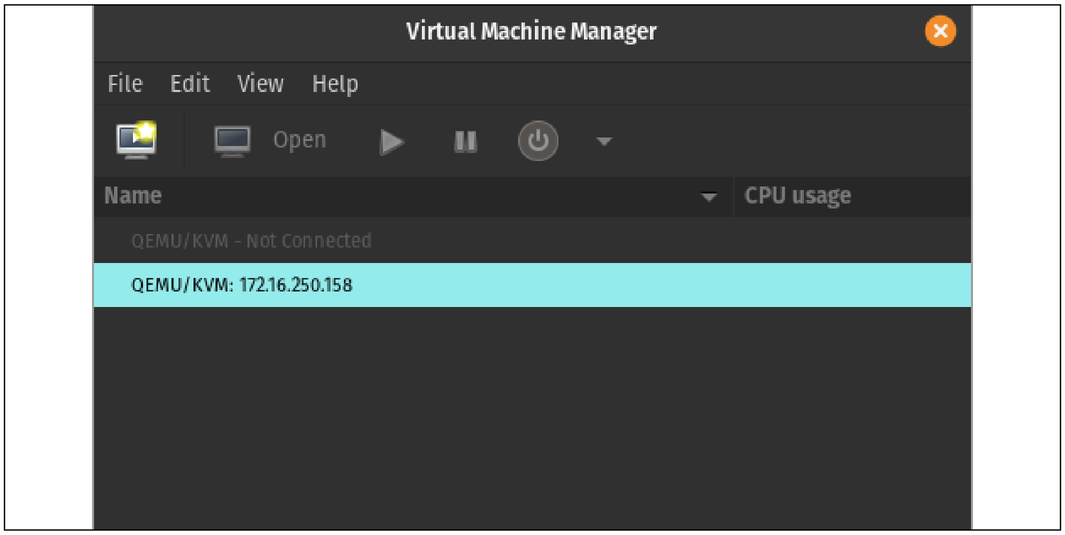


Hypervisor: QEMU/KVM
☑ Connect to remote host over SSH
Username: jay
Hostname: 172.16.250.158
Autoconnect: ☐
Generated URI: qemu+ssh://jay@172.16....

Figure 16.5: Adding a new connection to virt-manager

In the **Add Connection** window, enter the details for your connection. In the screenshot, you can see that I first checked the **Connect to remote host over SSH** box, which selects SSH as my connection method, `jay` for my **Username**, and I've entered the IP address of my KVM server (`172.16.250.158`) in the **Hostname** field. Fill out the specific details here for your KVM server to set up your connection. Keep in mind that in order for this to work, the username you include here will need to be able to access the server via SSH and have permissions to the hypervisor (be a member of the `kvm` and `libvirtd` groups we added earlier), and the `libvirtd` service must be running on the server. If all of these requirements are met, you'll have a new connection set up to your KVM server when you click **Connect**. You might see a pop-up dialog box with the text `Are you sure you wish to continue connecting (yes/no)?`. If you do, type `yes` and press *Enter*.

Either way, you should be prompted for your password to your KVM server; type that in and press *Enter*. You should now have a connection listed in your `virt-manager` application. You can see the connection I added in the following screenshot; it's the second one on the list. The first connection is localhost, since I also have KVM running on my local laptop in addition to having it installed on a remote server:

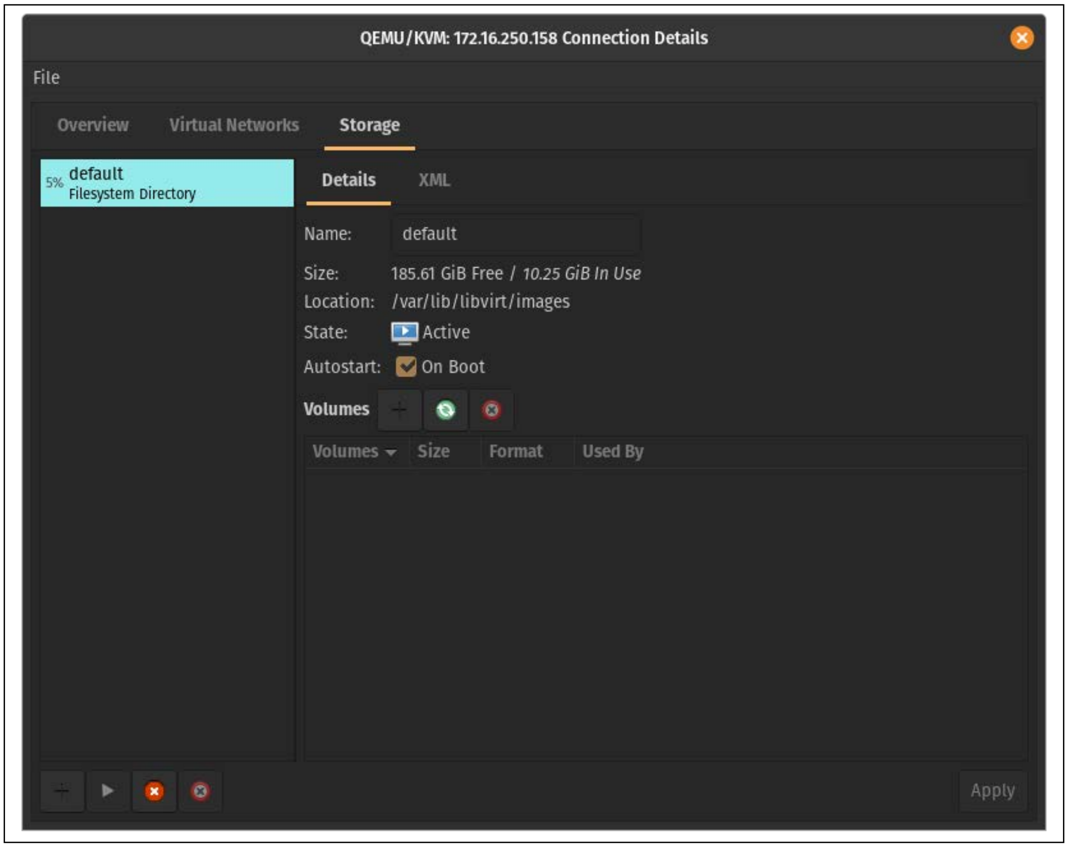


QEMU/KVM - Not Connected

QEMU/KVM: 172.16.250.158

Figure 16.6: virt-manager with a new connection added

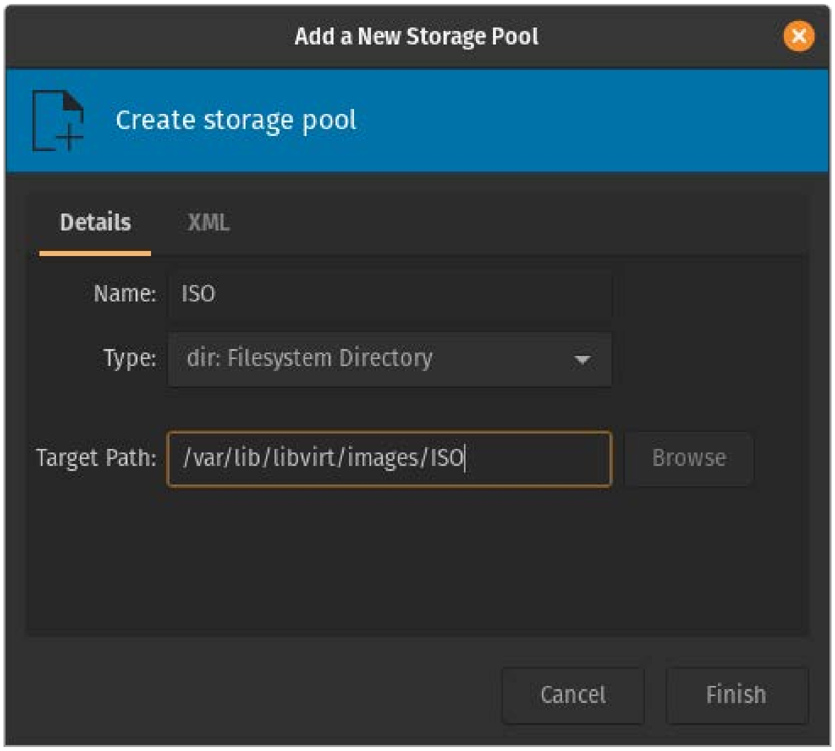
We're almost at a point where we'll be able to test our KVM server. But first, we'll need a storage group for ISO images, for use when installing operating systems on our VMs. When we create a VM, we can attach an ISO image from our ISO storage group to our VM, which will allow it to install the operating system. To create this storage group, open `virt-manager` if it's not open already. Right-click on the listing for your server connection and then click on **Details**. You'll see a new window that will show details regarding your KVM server. Click on the **Storage** tab:



Name: default
Size: 185.61 GiB Free / *10.25 GiB In Use*
Location: /var/lib/libvirt/images
State: Active
Autostart: On Boot

Figure 16.7: The first screen while setting up a new storage pool

At first, you'll only see the default connection we edited earlier. Now, we can add our ISO storage pool. Click on the plus symbol in the bottom-left corner to create the new pool:



Name: ISO
Type: dir: Filesystem Directory
Target Path: /var/lib/libvirt/images/ISO

Figure 16.8: The storage tab of the virt-manager application

In the **Name** field, type `ISO`. You can actually name it anything you want, but ISO makes sense, considering it will be storing ISO images. For the **Target Path** field, set it to `/var/lib/libvirt/images/ISO` unless you have a different directory in your filesystem for VM storage. Flick **Finish** to finalize our changes. We should also update the permissions for this directory so that it's owned by the proper user, and members of the `kvm` group have read and write access to it:

```
sudo chown root:kvm /var/lib/libvirt/images/ISO
sudo chmod g+rw /var/lib/libvirt/images/ISO
```

Congratulations! You now have a fully configured KVM server for creating and managing VMs. Our server has a place to store VMs as well as ISO images. You should also be able to connect to this instance using `virt-manager`, as we've done in this section. Next, I'll walk you through the process of setting up your first VM. Before we get to that, I recommend you copy some ISO images over to your KVM server. It doesn't really matter which ISO image you use, any operating system should suffice. If in doubt, you can simply download Ubuntu Server 20.04 again like we did back in *Chapter 1*, *Deploying Ubuntu Server*, when we set up our initial installation.

After you've chosen an ISO file and you've downloaded it, copy it over to your server via `scp` or `rsync`, and move it into the `/var/lib/libvirt/images/ISO` directory. Both of those utilities were covered in *Chapter 12*, *Sharing and Transferring Files*. Once the file has been copied over, you should have everything you need for now.

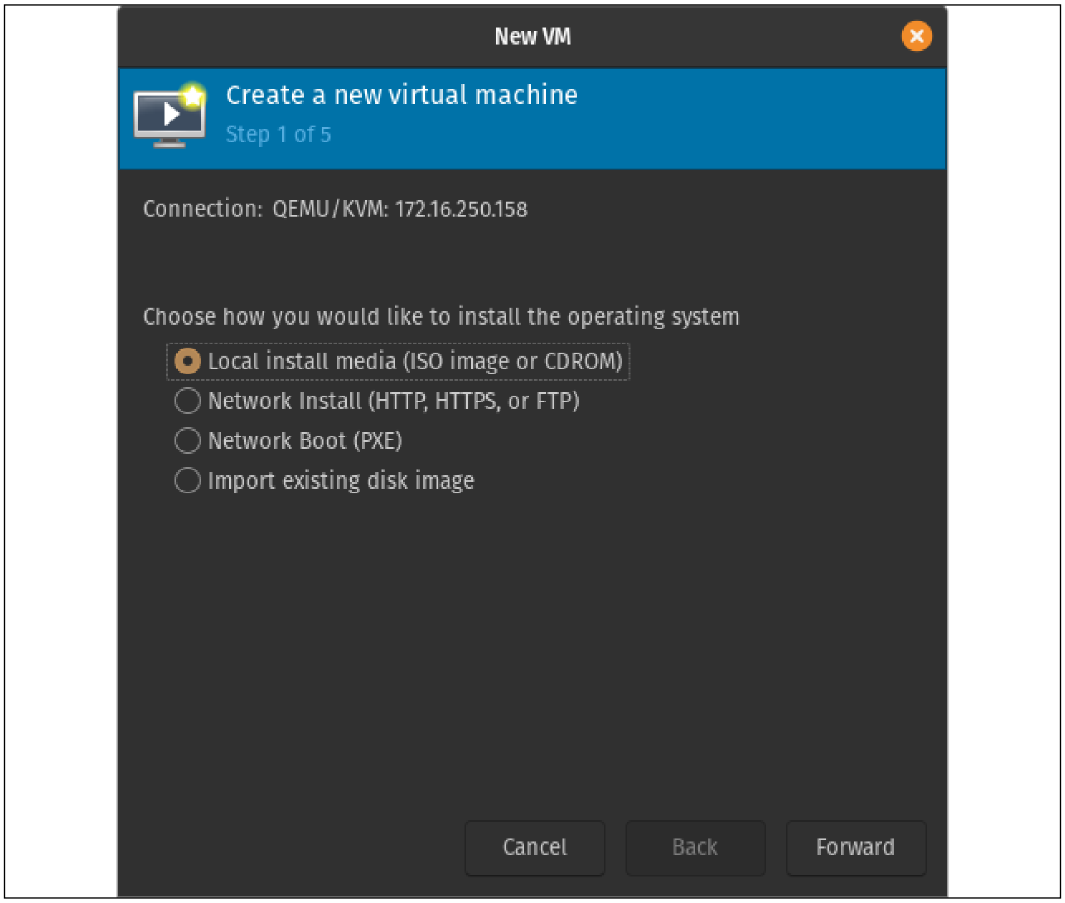
## Creating virtual machines

Now, the time has come to put your new VM server to the test and create a VM. At this point, I'm assuming that the following is true:

- You're able to connect to your KVM server via `virt-manager`
- You've already copied one or more ISO images to the server
- Your storage directory has at least 10 GB of space available
- The KVM server has enough free RAM to be associated with the VM you intend on creating

Go ahead and open up `virt-manager`, and let's get started!

In `virt-manager`, right-click your server connection and click on **New** to start the process of creating a new VM. The default selection will be on **Local install media (ISO image or CDROM)**; leave this selection and click on **Forward**:



Connection: QEMU/KVM: 172.16.250.158

Choose how you would like to install the operating system

- [✓ Local install media (ISO image or CDROM)]
- Network Install (HTTP, HTTPS, or FTP)
- Network Boot (PXE)
- Import existing disk image

Figure 16.9: The first screen while setting up a new VM

On the next screen, click on **Browse** to open up another window where you can select an ISO image you've downloaded:

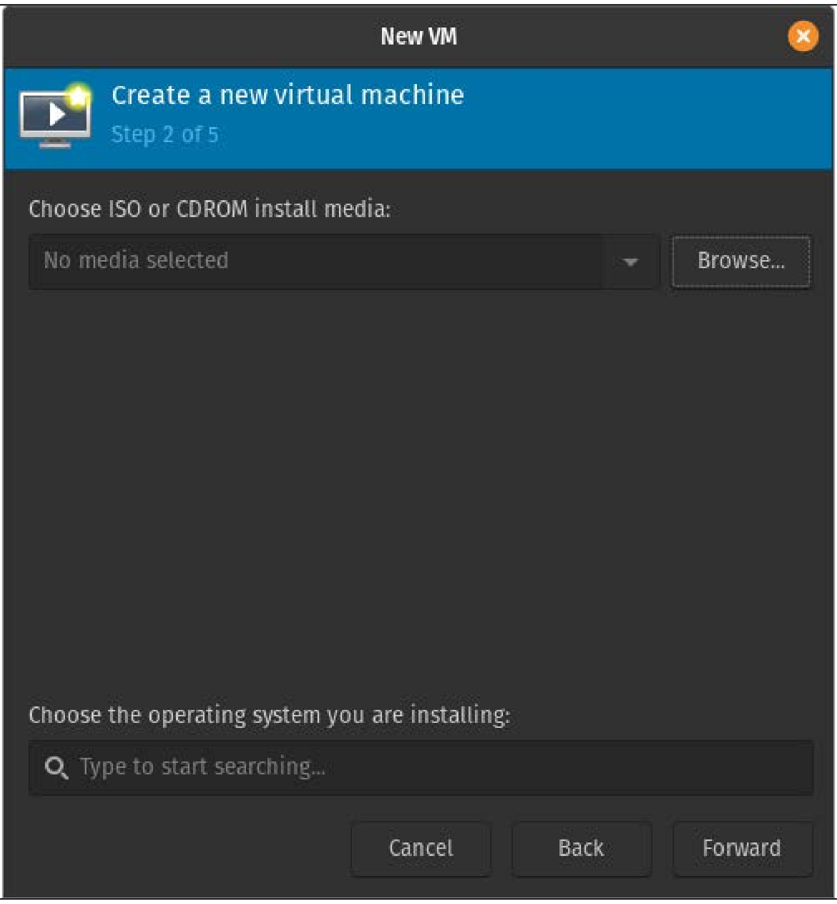
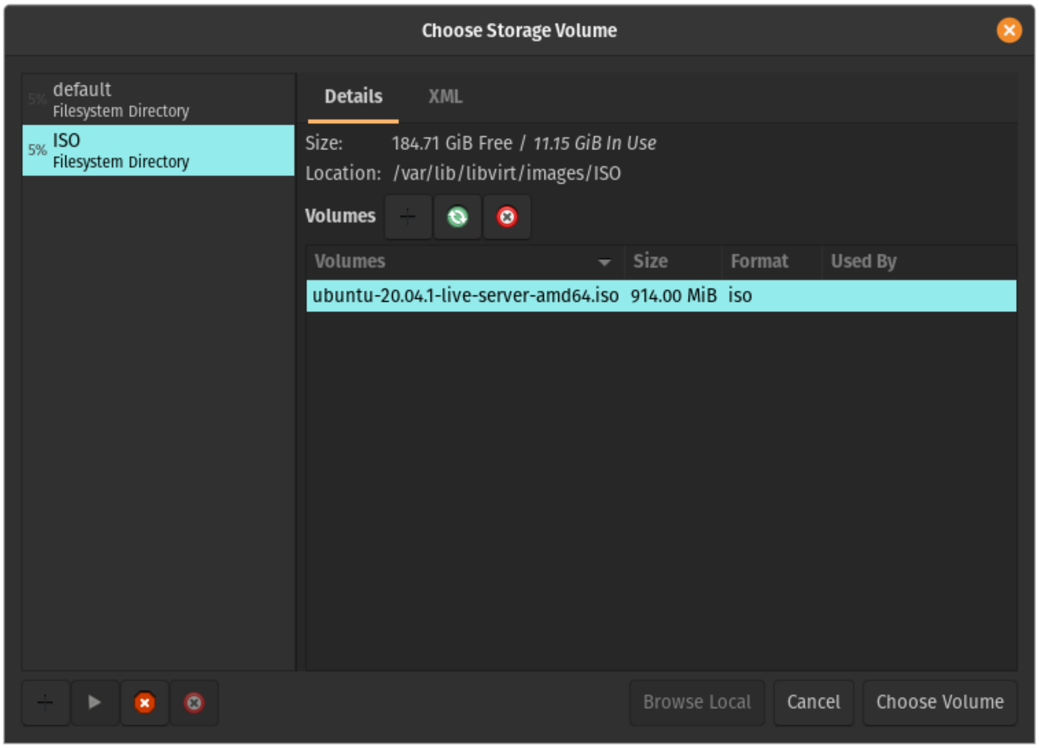

New VM
Create a new virtual machine
Step 2 of 5
Choose ISO or CDROM install media:
No media selected
Browse...
Choose the operating system you are installing:
Type to start searching...
Cancel
Back
Forward

Figure 16.10: Creating a new VM and setting the VM options

If you click on your ISO storage pool, you should see a list of ISO images you've downloaded:



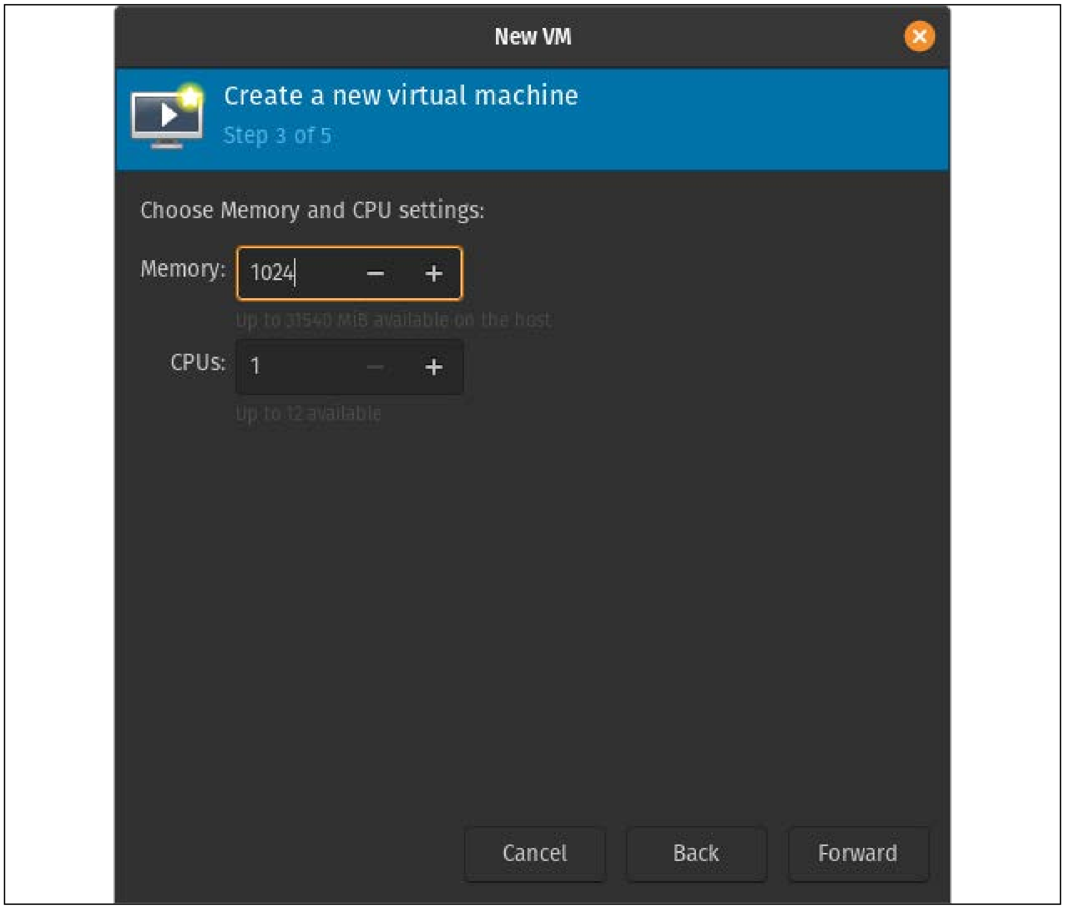
Size: 184.71 GiB Free / 11.15 GiB In Use
Location: /var/lib/libvirt/images/ISO

| Volumes | Size | Format | Used By |
|---|---|---|---|
| ubuntu-20.04.1-live-server-amd64.iso | 914.00 MiB | iso | |

Figure 16.11: Choosing an ISO image during VM creation

If you don't see any ISO images here, you may need to click the refresh icon. In my sample server, I added an install image for Ubuntu Server 20.04, which you can see in the list. Again, you can use whatever operating system you prefer. Click on the ISO image name to highlight it, and then click **Choose Volume** to finalize the selection. Then, click **Forward** to continue to the next screen.

Next, you'll be asked to allocate RAM and CPU resources to the VM:



Choose Memory and CPU settings:

Memory: 1024

Up to 31540 MiB available on the host

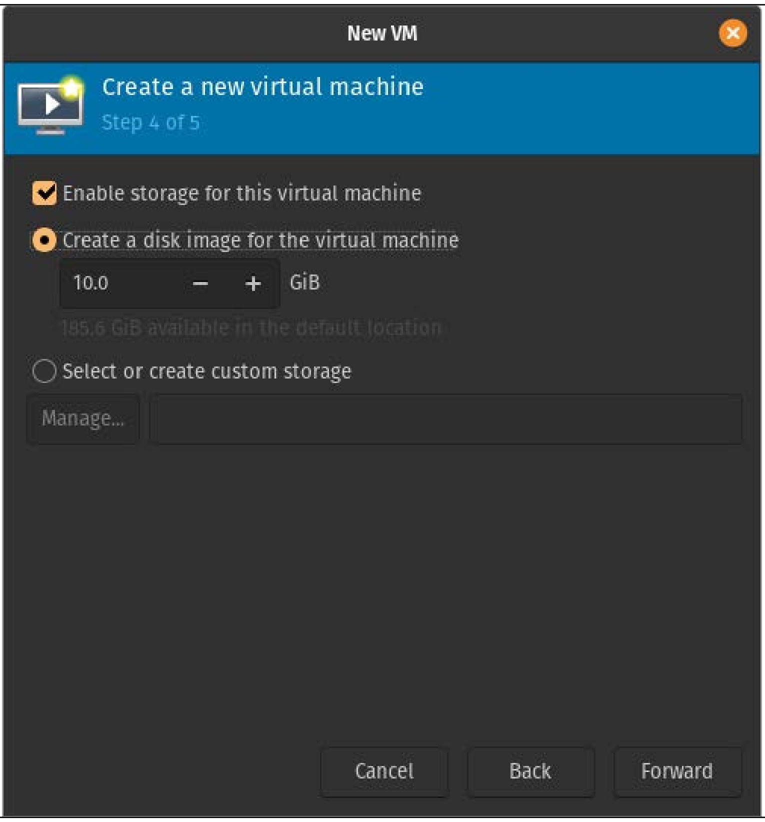
CPUs: 1

Up to 12 available

Figure 16.12: Adjusting RAM and CPU count for the new VM

For most Linux distributions with no GUI, 1,024 MB is plenty (unless your workload demands more). One CPU core is fine for lightweight workloads, but consider adding more if the documentation for the application you intend on running recommends more than that. The resources you select here will depend on what you have available on your host. Click on **Forward** when you've finished allocating resources.

Next, you'll allocate free disk space for your VM's virtual hard disk:



[✓] Enable storage for this virtual machine

[● Create a disk image for the virtual machine] 10.0 GiB

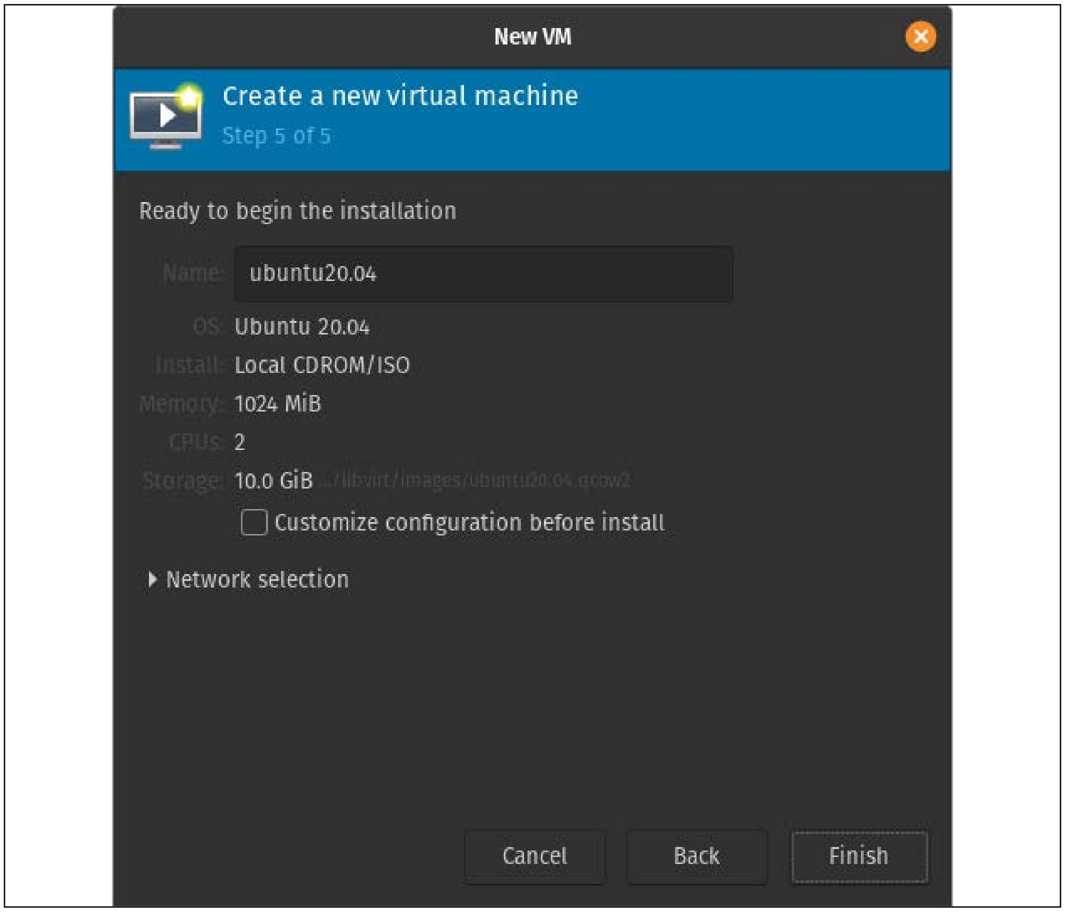
185.6 GiB available in the default location

○ Select or create custom storage

Figure 16.13: Allocating storage resources for the new VM

Set the disk image size to however much space you feel is relevant for the purpose of the VM. Click on **Forward** when done.

Finally, you'll name your VM:



New VM

Create a new virtual machine
Step 5 of 5

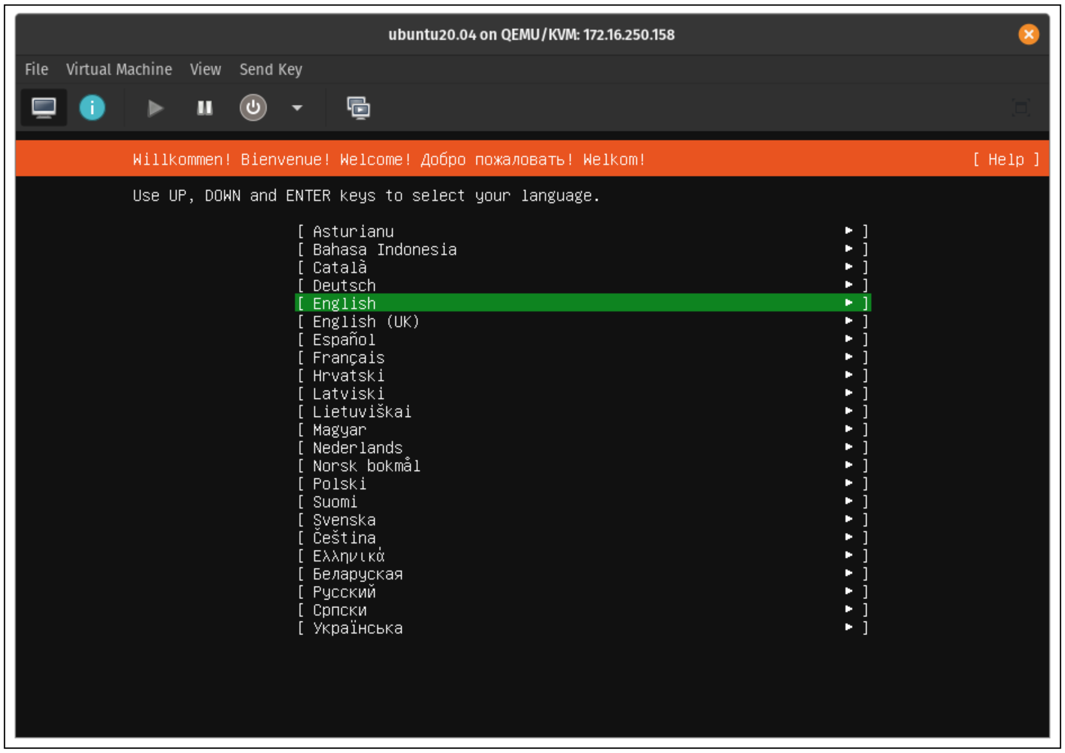
Ready to begin the installation

Name: ubuntu20.04
OS: Ubuntu 20.04
Install: Local CDROM/ISO
Memory: 1024 MiB
CPUs: 2
Storage: 10.0 GiB ../libvirt/images/ubuntu20.04.qcow2

☐ Customize configuration before install

Figure 16.14: Naming the new VM

This won't be the hostname of the VM; it's just the name you'll see when you see the VM listed in `virt-manager`. When you click on **Finish**, the VM will start and it will automatically boot into the install ISO you've attached to the VM near the beginning of the process. The installation process for that operating system will then begin:



```
Willkommen! Bienvenue! Welcome! Добро пожаловать! Welkom!                [ Help ]

Use UP, DOWN and ENTER keys to select your language.

                    [ Asturianu                                         ▸ ]
                    [ Bahasa Indonesia                                  ▸ ]
                    [ Català                                            ▸ ]
                    [ Deutsch                                           ▸ ]
                    [ English                                           ▸ ]
                    [ English (UK)                                      ▸ ]
                    [ Español                                           ▸ ]
                    [ Français                                          ▸ ]
                    [ Hrvatski                                          ▸ ]
                    [ Latviski                                          ▸ ]
                    [ Lietuviškai                                       ▸ ]
                    [ Magyar                                            ▸ ]
                    [ Nederlands                                        ▸ ]
                    [ Norsk bokmål                                      ▸ ]
                    [ Polski                                            ▸ ]
                    [ Suomi                                             ▸ ]
                    [ Svenska                                           ▸ ]
                    [ Čeština                                           ▸ ]
                    [ Ελληνικά                                          ▸ ]
                    [ Беларуская                                        ▸ ]
                    [ Русский                                           ▸ ]
                    [ Српски                                            ▸ ]
                    [ Українська                                        ▸ ]
```

Figure 16.15: Installing Ubuntu Server inside a VM

When you click on the VM window, it will steal your keyboard and mouse and dedicate it to the window. Press *Ctrl* and *Alt* at the same time to release this control and regain full control of your keyboard and mouse.

Unfortunately, I can't walk you through the installation process of your VM's operating system, since there are hundreds of possible candidates you may be installing. If you're installing another instance of Ubuntu Server, you can refer back to *Chapter 1*, *Deploying Ubuntu Server*, where we walked through the process. The process will be the same in the VM. From here, you should be able to create as many VMs as you need and have resources for.

Next, we'll look at some concepts surrounding networking for our VMs.

## Bridging the virtual machine network

Your KVM VMs will use their own network, unless you configure bridged networking. This means your VMs will get an IP address in their own network, instead of yours. By default, each machine will be a member of the `192.168.122.0/24` network, with an IP address in the range of `192.168.122.2` to `192.168.122.254`. If you're utilizing KVM VMs on your personal laptop or desktop, this behavior might be adequate. You'll be able to SSH into your VMs via their IP addresses if you're connecting from the same machine the VMs are running on. If this satisfies your use case, there's no further configuration you'll need to do.

Bridged networking allows your VMs to receive an IP address from the DHCP server on your network instead of its internal one, which will allow you to communicate with your VMs from any other machine on your network. This use case is preferable if you're setting up a central VM server to power infrastructure for your small office or organization, as your DHCP server can become a single source of truth for all of the IP addresses in use in your organization. With a bridged network on your VM server, each VM will be treated as any other network device. All you'll need is a wired network interface, as wireless cards typically don't work with bridged networking.

> That last point is very important. Some network cards don't support bridging, and if yours doesn't, you won't be able to use a bridge with your VM server unless you replace the network card. Before continuing, you may want to ensure your network card supports bridging by reading the documentation from the vendor of your device. In my experience, most wired cards made by Intel support bridging, and most wireless cards do NOT. Make sure you back up the Netplan configuration file before changing it, so you can revert back to the original version if you find that bridging doesn't work for you.

To set up bridged networking, we'll need to create a new interface on our server. Open up the `/etc/netplan/00-installer-config.yaml` file in your text editor with `sudo`. We already talked about this file in *Chapter 10, Connecting to Networks*, so I won't go into too much detail about it here. Basically, this file includes configuration for each of our network interfaces, and this is where we'll add our new bridged interface.

Make sure you make a backup of the original Netplan configuration file, and then replace its contents with the following. Be sure to replace `enp0s3` (the interface name) with your actual wired interface name if it's different. There are two occurrences of it in the file.

If you're reading the digital version of this book, it's highly recommended that you refrain from copying and pasting the following code, but rather type it manually or copy it from the GitHub URL for the book's code bundle. The reason being, the YAML format is extremely picky about spaces, and if you end up with a mix of spaces and tabs, the file might not work. When Netplan errors, it can be very hard to figure out exactly what it's complaining about, but spacing is quite often the culprit even if the error output doesn't lead you to believe so.

Take your time while configuring this file. If you make a single mistake, you will likely not have network access to the machine once it restarts:

```
network:
  ethernets:
    enp0s3:
      dhcp4: false
  bridges:
    br0:
      interfaces: [enp0s3]
      dhcp4: true
      parameters:
        stp: false
        forward-delay: 0
```

After you make the change, you can apply the new settings immediately, or simply reboot the server. If you have a monitor and keyboard hooked up to the server, the following command is the easiest way to activate the new configuration:

```
sudo netplan apply
```

If you're connected to the server via SSH, restarting networking will likely result in the server becoming inaccessible because the SSH connection will likely drop as soon as the network stops. This will disrupt the connection and prevent networking from starting back up. If you know how to use `screen` or `tmux`, you can run the `restart` command from within either; otherwise, it may just be simpler for you to reboot the server.

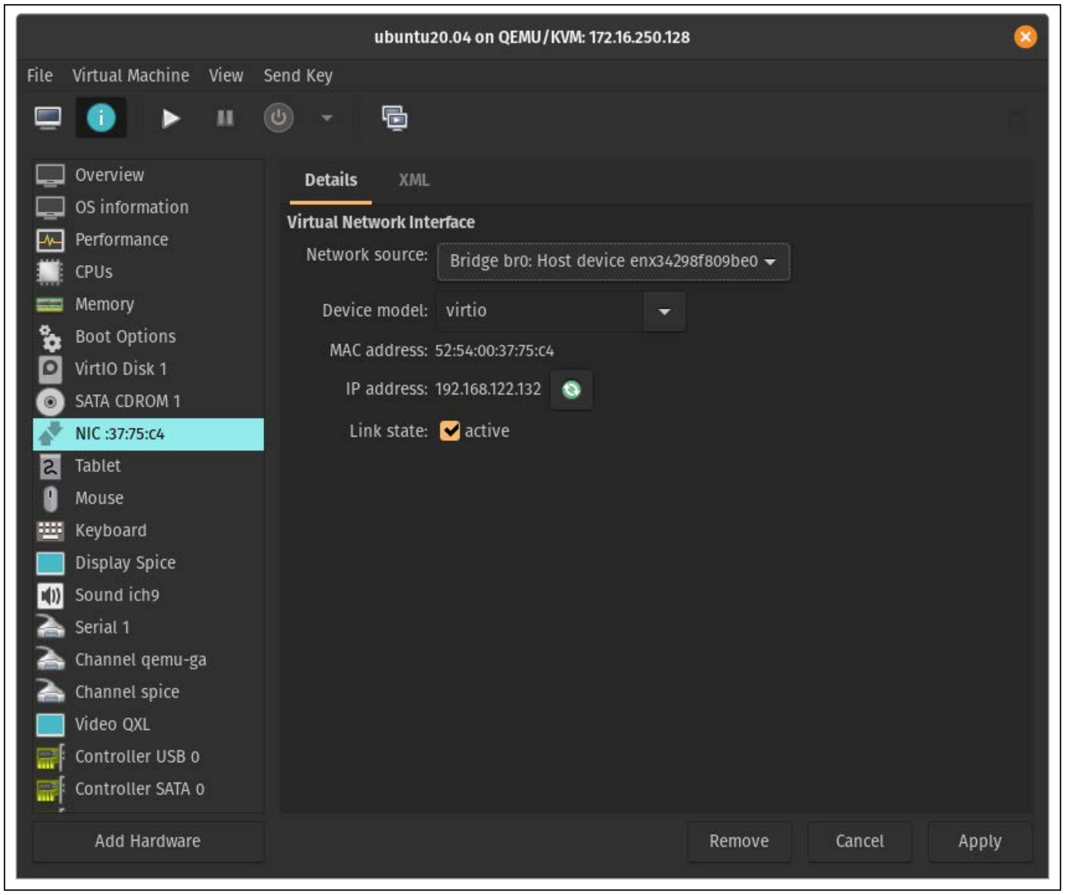
After networking restarts or the server reboots, check whether you can still access network resources, such as pinging websites and accessing other network nodes from it. If you can, you're all set. If you're having any trouble, make sure you edited the `/etc/netplan/00-installer-config.yaml` file properly.

Now, you should see an additional network interface listed when you run `ip addr show`. The interface will be called `br0`. The `br0` interface should have an IP address from your DHCP server, in place of your `enp0s3` interface (or whatever it may be named on your system). From this point forward, you'll be able to use `br0` for your VM's networking, instead of the internal network. The internal KVM network will still be available, but you can select `br0` to be used instead when you create new VMs.

If you have a VM you've already created that you'd like to switch to utilize your bridged networking, you can use the following steps to convert it:

1. First, open `virt-manager` and double-click on your VM. A new window with a graphical console of your VM will open.
2. The second button along the top (that appears as a blue circle) will open the **Virtual Hardware Details** tab, which will allow you to configure many different settings for the VM, such as the CPU count, RAM amount, boot device order, and more.
3. Among the options on the left-hand side of the screen, there will be one that reads **NIC** and shows part of the VM's network card's MAC address. If you click on this, you can configure the VM to use your new bridge by selecting it in the list.

4. Finally, click on **Apply**. You may have to restart the VM for the changes to take effect:



Virtual Network Interface

Network source: Bridge br0: Host device enx34298f809be0

Device model: virtio

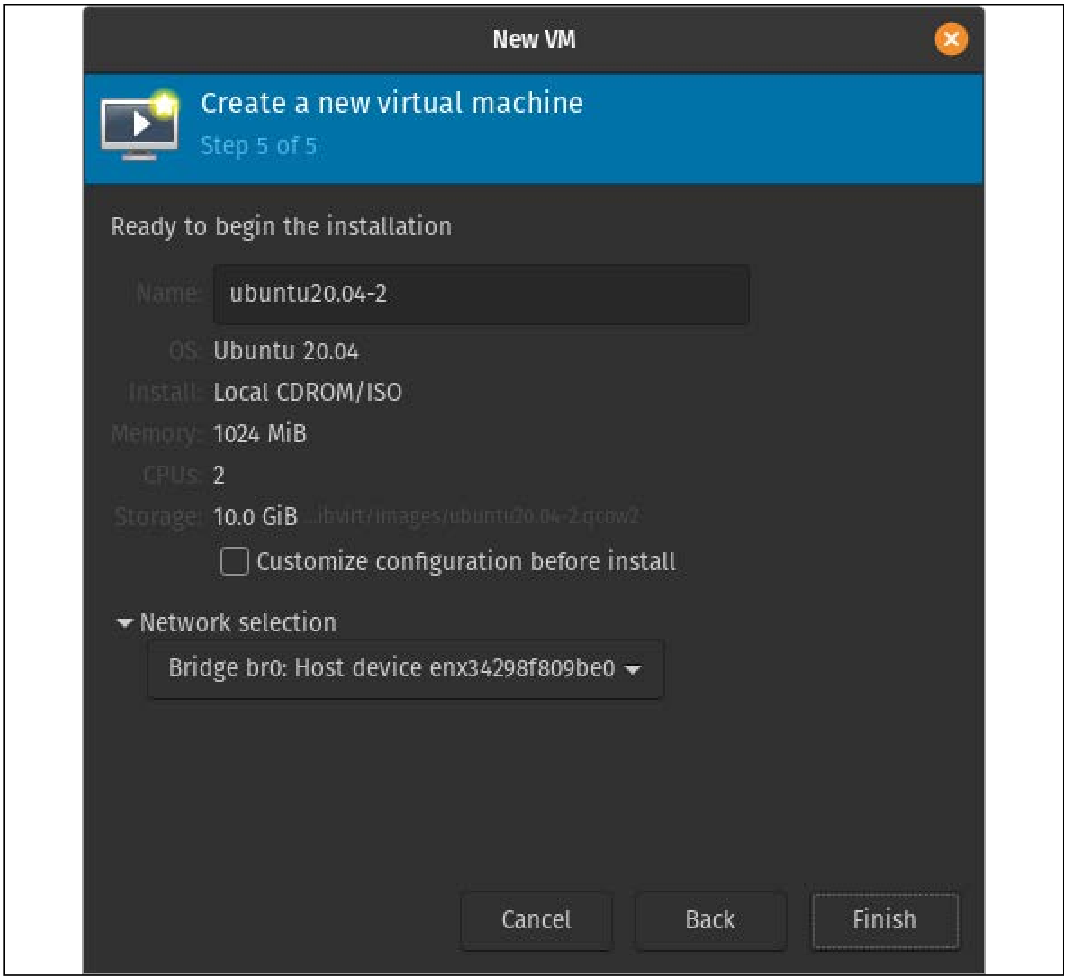
MAC address: 52:54:00:37:75:c4

IP address: 192.168.122.132

Link state: ✓ active

Figure 16.16: Configuring a VM to use bridge br0

While creating a brand-new VM, there's an additional step you'll need to do in order to configure the VM to use bridged networking. On the last step of the process, where you set a name for the VM (shown in *Figure 16.14*), you'll also see **Advanced options** listed near the bottom of the window. Expand this, and you'll be able to set your network name. Change the dropdown in this section to **Specify shared device name** and set the bridge **Name** to `br0`. Now, you can click on **Finish** to finalize the VM as before, and it should use your bridge whenever it starts up:


New VM

Create a new virtual machine
Step 5 of 5

Ready to begin the installation

Name: ubuntu20.04-2
OS: Ubuntu 20.04
Install: Local CDROM/ISO
Memory: 1024 MiB
CPUs: 2
Storage: 10.0 GiB ...ibvirt/images/ubuntu20.04-2.qcow2

☐ Customize configuration before install

Network selection
Bridge br0: Host device enx34298f809be0

Figure 16.17: Selecting a bridge for a newly created VM

From this point onward, you should have not only a fully-configured KVM server or instance, but also a solution that can be treated as a full citizen of your network. Your VMs will be able to receive an IP address from a DHCP server and communicate with other network nodes directly. If you have a very beefy KVM server, you may even be able to consolidate other network appliances into VMs to save space, which is basically the entire purpose of virtualization.

In the next section, we'll simplify the process a bit by discussing the creation of a template that can be used to act as a pre-configured starting point when setting up a new VM.

## Simplifying virtual machine creation with cloning

Now that we have a KVM server, and we can spin up an army of VMs to do our bidding, we can try and find clever ways of automating some of the workload of setting up a new VM. Every time we go to create a new VM, we need to go through the entire installation process for its operating system again. While this process is not difficult, we can certainly simplify it.

Most prominent virtualization solutions include a feature that allows you to create a **VM Template**. With a template, we can create a VM once and get it completely configured. Then, we can convert it to a template and use it as a base for all future VMs that will use that same operating system. This saves a tremendous amount of time. You'll probably recall the handful of screens you had to navigate through to install Ubuntu Server in our first chapter. Imagine not having to go through that process again (or at least not nearly as often).

Unfortunately, as great as QEMU/KVM is, it doesn't have a template feature. This glaring hole in its feature set is a sizable setback, but thankfully we Linux administrators are very clever, and we can easily work around this to create a solution that's essentially the same thing as templates.

Take the following screenshot, for example:

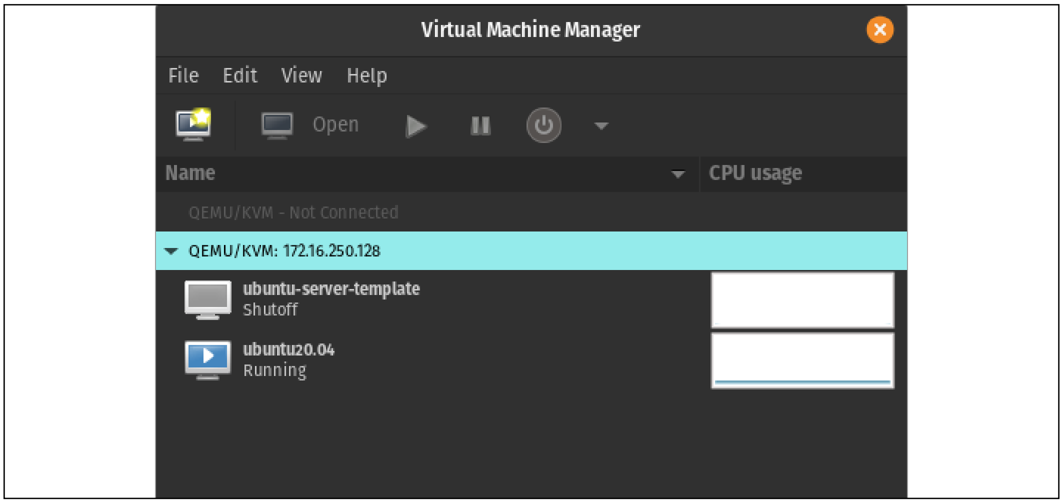
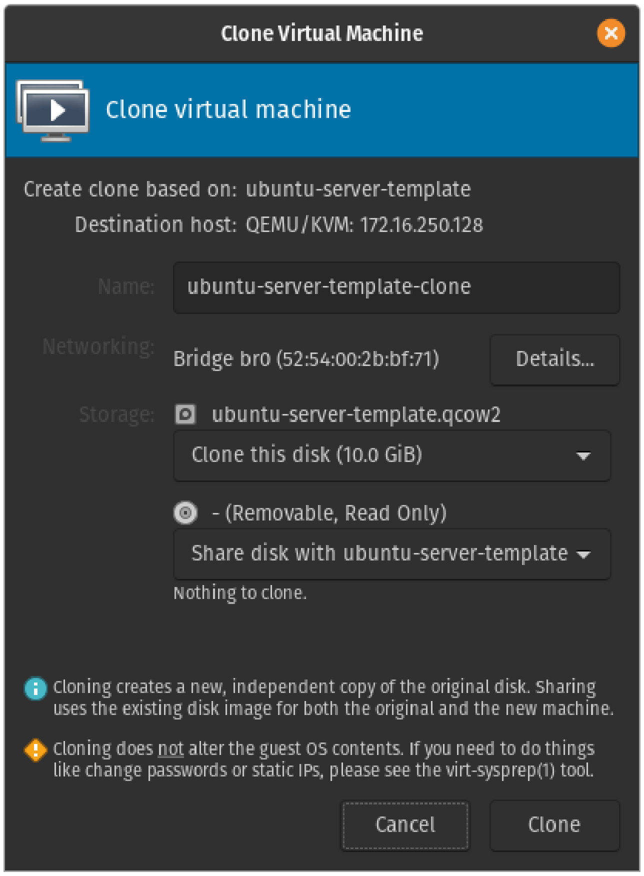


Figure 16.18: Selecting a bridge for a newly created VM

In the screenshot, you can see two VMs, `ubuntu20.04` and `ubuntu-server-template`. Although its name would lead you to believe otherwise, the latter is not a template at all; it's just a VM. There's nothing really different about it, aside from the fact that it isn't running. What it is, though, is a clever workaround (if I do say so myself). If I want to create a new VM, I simply right-click on it, then click **Clone**.

The following window will appear:



Create clone based on: ubuntu-server-template
Destination host: QEMU/KVM: 172.16.250.128

Name: ubuntu-server-template-clone

Networking: Bridge br0 (52:54:00:2b:bf:71) Details...

Storage: ubuntu-server-template.qcow2
Clone this disk (10.0 GiB)

- (Removable, Read Only)
Share disk with ubuntu-server-template
Nothing to clone.

Cloning creates a new, independent copy of the original disk. Sharing uses the existing disk image for both the original and the new machine.

Cloning does not alter the guest OS contents. If you need to do things like change passwords or static IPs, please see the virt-sysprep(1) tool.

Figure 16.19: Cloning a VM

When I click **Clone** in this window, after giving the new VM a name, I've made a copy of it to serve as my new VM. It will use the original as a base, which I've already configured. Since Ubuntu Server was installed on the "template," I don't need to do all that work again.

Think about the tasks that you find yourself doing manually after setting up a new Ubuntu Server instance. With a base VM being used as if it were a template, you can include any tweaks or customizations you find yourself implementing right into that VM, so every time you clone it, all that work is done for you automatically. So long as you maintain your base VM, you can spin up as many VMs from it as you need and be able to do so with minimal configuration steps.

We've used `virt-manager` quite a bit in this chapter to customize our VMs, and while it's a great utility, we should also understand how to manage our infrastructure without it. In the next section, we'll take a look at some command-line examples of managing VMs.

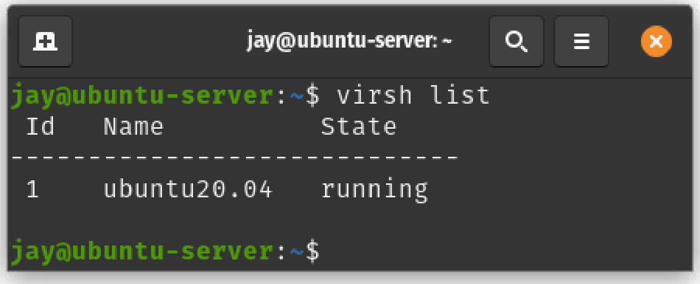
## Managing virtual machines via the command line

In this chapter, I showed you how to manage VMs with `virt-manager`. This is great if you have a secondary machine with a GUI running Linux as its operating system. But what do you do if such a machine isn't available, and you'd like to perform simple tasks such as rebooting a VM or checking to see which VMs are running on the server?

On the VM server itself, you have access to the `virsh` suite of commands, which will allow you to manage VMs even if a GUI isn't available. To use these commands, simply connect to the machine that stores your VMs via SSH. What follows are some easy examples to get you started. Here's the first one:

```
virsh list
```

This command will return an output like that shown in the following screenshot:



```
jay@ubuntu-server:~$ virsh list
 Id   Name          State
------------------------------
 1    ubuntu20.04   running

jay@ubuntu-server:~$
```

Figure 16.20: Showing running VMs with the virsh list command

With one command, we were able to list the VMs running on the server. In the example screenshot, you can see that I have a single VM running. If you'd also like to see non-running instances, simply add the `--all` option to the command.

We can manage the state of our VMs with any of the following commands:

```
virsh start vm-name
virsh shutdown vm-name
virsh suspend vm-name
virsh resume vm-name
virsh destroy vm-name
virsh undefine vm-name
```

The command syntax for `virsh` is extremely straightforward. By looking at the previous list of commands, you should be able to glean exactly what they do. The `virsh` commands allow us to do things such as `start`, `shutdown`, `suspend`, and `resume` a VM. The `virsh destroy` command is potentially destructive, as we'd use it when we want to halt a VM abruptly. It's essentially the same result as pulling a power cable from a physical server; it stops the instance immediately. You should only run that command when you are dealing with an unresponsive VM. Finally, the `virsh undefine` command deletes a VM, but you'll have to remove any associated disk files with the `rm` command. The default directory for disk files is `/var/lib/libvirt/images`, so you can look inside that directory for any disk files that belong to the VM you've deleted (they will be named the same as the VM).

That's not all `virsh` can do, however. We can actually create a VM with the `virsh` suite of commands as well. Learning how to do so is a good idea if you don't use Linux as your workstation operating system, or you don't have access to `virt-manager` for some reason. However, manually creating VM disk images and configuration is outside the scope of this chapter. The main goal is for you to familiarize yourself with managing VMs via `virsh`, and these simple basics will allow you to expand your knowledge further.

## Summary

In this chapter, we took a look at virtualization, specifically with QEMU/KVM. We walked through the installation of KVM and the configuration required to get our virtualization server up and running. We walked through the process of creating a bridged network so that our VMs can be accessible from the rest of the network and created our first VM. In addition, although QEMU/KVM doesn't have its own solution for templating, we worked around that and created our own solution.

In *Chapter 17*, *Running Containers*, we'll take a look at containerization, which will include both Docker and LXD. Stay tuned!

## Further reading

- Ubuntu `virsh` documentation: `https://help.ubuntu.com/community/KVM/Virsh`
- *Mastering KVM Virtualization* (Packt Publishing): `https://www.packtpub.com/networking-and-servers/mastering-kvm-virtualization`

# 17
# Running Containers

The IT industry never ceases to amaze me. Back when the concept of virtualization came about, it revolutionized the data center. Virtualization allowed us to run many small **Virtual Machines** (**VMs**) on one server, effectively allowing us to consolidate the equipment in our server racks. And just when we thought it couldn't get any better, the concept of containerization took the IT world by storm, allowing us to build portable instances of our software that not only improved how we deploy applications but also changed the way we develop them. In this chapter, we will cover the exciting world of containerization. This exploration will include:

- What is containerization?
- Understanding the differences between Docker and LXD
- Installing Docker
- Managing Docker containers
- Automating Docker image creation with Dockerfiles
- Managing LXD containers

To begin, let's explore what containerization is, how it differs from virtualization, and some considerations around how such technology might be implemented.

## What is containerization?

In the last chapter, we covered virtualization. Virtualization allows us to run multiple *virtual* servers on one physical piece of hardware. We allocate CPU, RAM, and disk space to these VMs, and they run as if they were a real server. In fact, for all intents and purposes, a VM is a real server.

However, there are also weaknesses with VMs. Perhaps the most glaringly obvious is the resources you allocate to a VM, which are likely being wasted. For example, perhaps you've allocated 512 MB of RAM to a VM. What if the application only rarely uses more than 100 MB of RAM? That means most of the time, 412 MB of RAM that could otherwise be used for a useful purpose is just sitting idle. The same can be said of CPU usage. Nowadays, VM solutions do have ways of sharing unused resources, but effectively, resource efficiency is a natural weakness of the platform.

Containers, unlike VMs, are not actual servers. At least, not in the way you typically think about them. While VMs typically have a dedicated CPU, containers share the CPU with the host. VMs also generally have their own kernel, but containers share the kernel of the host. Containers are still segregated, though. Just as a VM cannot access the host filesystem, a container can't either (unless you explicitly set it up to do so).

What is a container, then? It's probably best to think of a container as a filesystem rather than a VM. The container itself contains a file structure that matches that of the distribution it's based on. A container based on Ubuntu Server, for example, will have the same filesystem layout as a real Ubuntu Server installation on a VM or physical hardware. Imagine copying all the files and folders from an Ubuntu installation, putting them all in a single segregated directory, and having the binary contents of the filesystem executed as a program, without an actual operating system running.

To be fair, that description was an oversimplification of how containers actually run on an Ubuntu server, as the technology utilizes the functionality of the Linux kernel to isolate various components of a container from the rest of the system. However, a full discussion of those technologies is beyond the scope of this book. But understanding that such isolation exists within containers is something you should keep in mind, as keeping processes that are running within a container separate from other processes running on the host server is an important benefit.

Portability is another strength of **containerization**. With a container, you can literally pass it around to various members of your development team, and then push the container into production when everyone agrees that it's ready. The container itself will run exactly the same on each workstation, regardless of which operating system the workstation uses. To be fair, you can export and import VMs on any number of hosts, but containers make this process extremely easy. In fact, portability is at the core of the design of this technology.

The concept of containerization is not necessarily new. When Docker hit the scene, it took the IT world by storm, but it was by no means the first solution to offer containerization. LXC, and other technologies, predate it. It was, however, a clever marketing tactic with a cool-sounding brand that launched containerization into mainstream popularity. By no means am I saying that Docker is all hype, though. It's an awesome technology with many benefits. It's definitely worth using, and you may even find yourself preferring it to VMs.

The main difference with containerization is that each container generally does one thing. For example, perhaps a container holds a hosted website or contains a single application. VMs are often created to do many tasks, such as a web server that hosts ten websites. Containers on the other hand are generally used for one task each, though depending on the implementation you may see others going against this norm.

When should you use containers? I recommend you consider containers any time you're running a web app or some sort of service and you'd benefit from sharing resources instead of dedicating memory or CPU. The truth is, not all applications will run well in a container, but it's at least something to consider. Any time you're running an application that is typically accessed via a web browser, it's probably better off in a container rather than a VM. As an administrator, you'll most likely experiment with the different tools available to you and decide the best tool for the job based on your findings.

Now that we understand the core concepts surrounding containers, let's explore the differences between two container technologies.

## Understanding the differences between Docker and LXD

In this chapter, we're going to explore both Docker and LXD and see examples of containers running in both. Before we start working on that though, it's a good idea to understand some of the things that set each solution apart from the other.

**Docker** is probably the technology most of my readers have heard of. It seems as though you can't visit a single IT conference nowadays without it at least being mentioned. Docker is everywhere, and it runs on pretty much any platform. There's lots of documentation available for Docker and various resources you can utilize to deploy it. Docker utilizes a *layered* approach to containerization. Every change you make to the container creates a new layer, and these layers can form the base of other containers, thus saving disk space. More on that later.

**LXD** (pronounced Lex-D) finds its roots in **LXC**, so it's important to understand that first before we talk about LXD. LXC (pronounced Lex-C) is short for **Linux Containers** and is another implementation of containerization, very similar to Docker. This technology uses the **control groups** (**cgroups**) feature of the Linux kernel, which isolates processes and is able to segregate them from one another. This enhances security, as processes should not be able to read data from other processes unless there's a good reason to.

LXC takes the concept of segregation even further, by creating an implementation of virtualization based solely on running applications in an isolated environment that matches the environment of an operating system. You can run LXC containers on just about every distribution of Linux available today.

LXD is also available for many Linux distributions, but it's treated as a first-class citizen in Ubuntu. This is because Canonical (the company behind Ubuntu) had a major hand in its development, and also offers commercial support for it. Since the software that makes LXD itself work is distributed via `snap` packages, that essentially means that any distribution of Linux that is able to install `snap` packages should be able to install LXD.

LXD takes LXC and gives it additional features that it otherwise wouldn't have, such as snapshots, ZFS support, and migration. LXD doesn't replace LXC; it actually utilizes it to provide its base technology. Perhaps the best way to think of LXD is as LXC with an additional management layer on top that adds additional features.

How does LXD/LXC differ from Docker? The main difference is that while they are both container solutions and solve the same goal in a very similar way, LXD is more similar to an actual VM while Docker tries harder to differentiate itself from that. In comparison, Docker containers are transactional (every task is run in a separate layer) and you generally have an `ENTRYPOINT` command that is run inside the container when you launch it. Essentially, LXD has a filesystem that you can directly access from the host operating system and has a simpler approach to containerization. You can think of LXC as a form of *machine* container that closely emulates a VM, and a Docker container as an *application* container that provides the foundation needed to run an application. Regardless of these differences, the technologies can be used in the same way and provide support for identical use cases.

When should you use Docker and when should you use LXD? I actually recommend you practice both, since they're not overly difficult to learn. We will go over the basics of these technologies in this chapter. But to answer the question at hand, there are a few use cases where one technology may make more sense than the other. Docker is more of a general-purpose tool. You can run Docker containers on Linux, macOS, and even Windows. It's therefore a good choice if you want to create a container that runs everywhere. LXD is generally best for Linux environments, though Docker runs great in Linux too. The operating system you're running your container solution on is of little importance nowadays, since most people use a container service to run containers rather than an actual server that you manage yourself. In the future, if you get heavily into containerization, you may find yourself forgoing the operating system altogether and just running them in a service such as Amazon's **Elastic Container Service** (**ECS**), which is one of a handful of cloud services that allow you to run containers without having to manage the underlying server.

Another benefit of Docker is **Docker Hub**, which you can use to download containers others have made or even upload your own for others to use. The benefit here is that if someone has already solved the goal you're trying to achieve, you can benefit from their work rather than starting from scratch, and they can also benefit from your work as well. This saves time and is often better than creating a solution by hand.

> Always make sure to audit third-party resources before you put them to use in your organization. This includes (but isn't limited to) containers developed by a third party.

Now that we understand not only the core concepts but the differences between two containerization standards, let's take a look at Docker.

## Installing Docker

Installing Docker is very fast and easy, so much so that it barely constitutes its own section. In the last chapter, we had to install several packages in order to get a **Kernel-based Virtual Machine** (**KVM**) virtualization server up and running as well as tweaking some configuration files. In comparison, installing Docker is effortless, as you only need to install the `docker.io` package:

```
sudo apt install docker.io
```

Yes, that's all there is to it. Installing Docker was definitely much easier than setting up KVM as we did in the previous chapter. Ubuntu includes Docker in its default repositories, so it's only a matter of installing this one package and its dependencies. You'll now have a new service installed on your machine, simply titled `docker`. In order to be useful, the service needs to be running. You can check to see whether or not it's already running with the following command:

```
systemctl status docker
```

Check the output of the previous command to see if the docker service is running. You should also check to see if the service is enabled. If not, you can start the service and also enable it at the same time with the following command:

```
systemctl enable --now docker
```

We also have the `docker` command available to us now, which allows us to manage our containers. By default, it does require `root` privileges, so you'll need to use `sudo` to use it. To make this easier, I recommend that you add your user account to the `docker` group before going any further. This will eliminate the need to use `sudo` every time you run a `docker` command. The following command will add your user account to the appropriate group:

```
sudo usermod -aG docker <yourusername>
```

After you log out and then log in again, you'll be able to manage Docker much more easily.

> You can verify your group membership by simply running the groups command with no options, which should now show you as a member of the `docker` group.

Well, that's it. Docker is installed, and your user account is a member of the `docker` group, so you're good to go. Wow, that was easy!

Now that we have Docker installed, let's start using it. It's a fun technology to learn, and in the next section we'll explore a few examples.

## Managing Docker containers

Now that Docker is installed and running, let's take it for a test drive. After installing Docker, we have the `docker` command available to use now, which has various sub-commands to perform different functions with containers. First, let's try out `docker search`:

```
docker search ubuntu
```

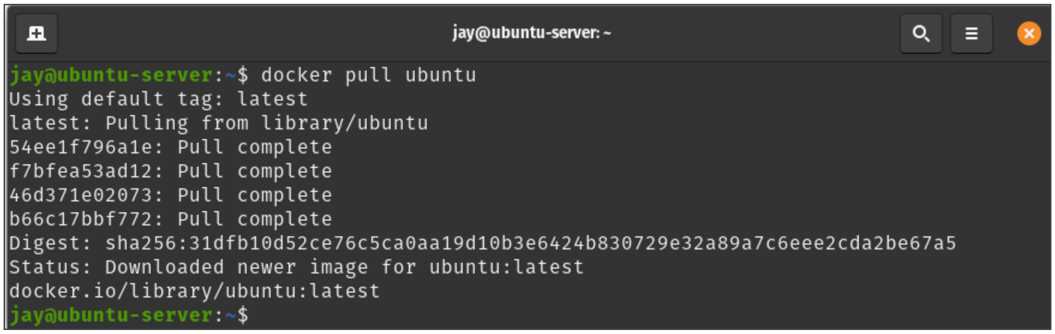
The `docker search` command allows us to search for a container given a search term. By default, it will search Docker Hub, which is an online repository that hosts containers for others to download and utilize. You could search for containers based on other distributions, such as Fedora or CentOS, if you wanted to experiment. The command will return a list of Docker images available that meet your search criteria.

So, what do we do with these images? An image in Docker is equivalent to a VM or hardware image. It's a snapshot that contains the filesystem of a particular operating system or Linux distribution, along with some changes the author included to make it perform a specific task. This image can then be downloaded and customized to suit your purposes. You can choose to upload your customized image back to Docker Hub, if you would like to contribute upstream. Every image you download will be stored on your machine so that you won't have to re-download it every time you wish to create a new container.

To pull down a Docker image for our use, we can use the `docker pull` command, along with one of the image names we saw in the output of our `search` command:

```
docker pull ubuntu
```

With the preceding command, we're pulling down the latest Ubuntu container image available on Docker Hub. The image will now be stored locally, and we'll be able to create new containers from it. The process will look similar to the following screenshot:



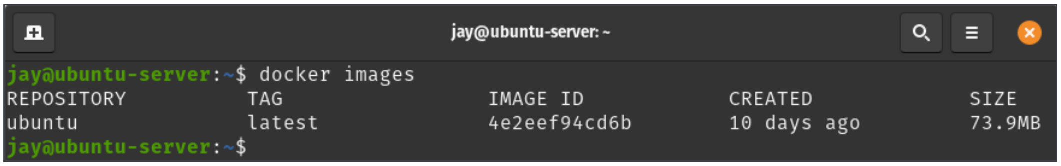
```
jay@ubuntu-server:~$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
54ee1f796a1e: Pull complete
f7bfea53ad12: Pull complete
46d371e02073: Pull complete
b66c17bbf772: Pull complete
Digest: sha256:31dfb10d52ce76c5ca0aa19d10b3e6424b830729e32a89a7c6eee2cda2be67a5
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
jay@ubuntu-server:~$
```

Figure 17.1: Downloading an Ubuntu container image

If you're curious as to which images you have saved locally, you can execute `docker images` to get a list of the Docker container images you have stored on your server:

```
docker images
```

The output will look similar to this:



```
jay@ubuntu-server:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              4e2eef94cd6b        10 days ago         73.9MB
jay@ubuntu-server:~$
```

Figure 17.2: Listing installed Docker images

Notice the `IMAGE ID` in the output. If for some reason you want to remove an image, you can do so with the `docker rmi` command, and you'll need to use the ID as an argument to tell the command what to delete. The syntax would look similar to this if I was removing the image with the ID shown in the screenshot:

```
docker rmi 4e2eef94cd6b
```

Feel free to remove the Ubuntu image if you've already downloaded it so you can see what the process looks like. You can always re-download it by running `docker pull ubuntu`.

Once you have a container image downloaded to your server, you can create a new container from it by running the `docker run` command, followed by the name of your image and an application within the image to run. An application run from within a Docker container is known as an `ENTRYPOINT`, which is just a fancy term to describe an application a particular container is configured to run. You're not limited to the `ENTRYPOINT` though, and not all containers actually have an `ENTRYPOINT`. You can use any command in the container that you would normally be able to run in that distribution. In the case of the Ubuntu container image we downloaded earlier, we can run `bash` with the following command so we can get a prompt and enter any command(s) we wish:

```
docker run -it ubuntu /bin/bash
```

Once you run that command, you're now interacting with a shell prompt from within your container. From here, you can run commands you would normally run within a real Ubuntu machine, such as installing new packages, changing configuration files, and more. Go ahead and play around with the container, and then we'll continue with a bit more theory on how this is actually working.

There are some potentially confusing aspects of Docker we should get out of the way first before we continue with additional examples. The thing that's most likely to confuse newcomers to Docker is how containers are created and destroyed. When you execute the `docker run` command against an image you've downloaded, you're actually creating a container. Therefore, the image you downloaded with the `docker pull` command wasn't an actual container itself, but it becomes a container when you run an instance of it. When the command that's being run inside the container finishes, the container goes away. Therefore, if you were to run `/bin/bash` in a container and install a bunch of packages, those packages would be wiped out as soon as you exited the container.

You can think of a Docker image as a "blueprint" for a container that can be used to create running containers. Every container you run has a container ID that differentiates it from others. If you want to remove a persistent container, for example, you would need to reference this ID with the `docker rm` command. This is very similar to the `docker rmi` command that's used to remove container images.

To see the container ID for yourself, you'll first need to exit the container if you're currently running one. There are two ways of doing so. First, you could press *Ctrl* + *d* to disconnect, or even type `exit` and press *Enter*. When you exit the container, you're removing it (Docker containers only typically exist while running). When you run the `docker ps` command (which is the command you'll use any time you want a list of containers on your system), you won't see it listed. Instead, you can add the `-a` option to see all containers listed, even those that have been stopped.

You're probably wondering, then, how to exit a container and not have it go away. To do so, while you're attached to a container, press *Ctrl* + *p* and then press *q* (don't let go of the *Ctrl* key while you press these two letters). This will drop you out of the container, and when you run the `docker ps` command (even without the `-a` option), you'll see that it's still running.

The `docker ps` command deserves some attention. The output will give you some very useful information about the containers on your server, including the `CONTAINER ID` that was mentioned earlier. In addition, the output will contain the `IMAGE` it was created from, the `COMMAND` being run when the container was `CREATED`, and its `STATUS`, as well as any `PORTS` you may have forwarded. The output will also display randomly generated names for each container, which are usually quite comical. As I was going through the process of creating containers while writing this section, the code names for my containers were `tender_cori`, `serene_mcnulty`, and `high_goldwasser`. This is just one of the many quirks of Docker, and some of these can be quite humorous.

The important output of the `docker ps -a` command is the `CONTAINER ID`, the `COMMAND`, and the `STATUS`. The ID, which we already discussed, allows you to reference a specific container to enable you to run commands against it. `COMMAND` lets you know what command was being run. In our example, we executed `/bin/bash` when we started our containers.

If we have any containers that were stopped, we can resume a container with the `docker start` command, giving it a container ID as an argument. Your command will end up looking similar to this:

```
docker start 353c6fe0be4d
```

The output will simply return the ID of the container, and then drop you back to your shell prompt—not the shell prompt of your container, but that of your server. You might be wondering at this point: how do I get back to the shell prompt for the container? We can use `docker attach` for that:

```
docker attach 353c6fe0be4d
```

The `docker attach` command is useful because it allows you to attach your shell to a container that is already running. Most of the time, containers are started automatically instead of starting with `/bin/bash` as we have done. If something were to go wrong, we may want to use something like `docker attach` to browse through the running container to look for error messages. It's very useful.

Speaking of useful, another great command is `docker info`. This command will give you information about your implementation of Docker, such as letting you know how many containers you have on your system, which should be the number of times you've run the `docker run` command unless you cleaned up previously run containers with `docker rm`. Feel free to take a look at its output and see what you can learn from it.

Getting deeper into the subject of containers, it's important to understand what a Docker container is and what it isn't. A container is not a service running in the background, at least not inherently. A container is a collection of namespaces, such as a namespace for its filesystem or users. As discussed earlier in this chapter, containers are isolated from the rest of the server by utilizing technology within the Linux kernel. When you disconnect without a process running within the container, there's no reason for it to run, since its namespace is empty. Thus, it stops. If you'd like to run a container in a way that is similar to a service (it keeps running in the background), you would want to run the container in **detached mode**. Basically, this is a way of telling your container to run this process and to not stop running it until you tell it to. Here's an example of creating a container and running it in detached mode:

```
docker run -dit ubuntu /bin/bash
```

After running the previous command, Docker will print a container ID, and then drop back to your command prompt. You can then see that the container is running with the `docker ps` command, and use `docker attach` along with the container ID to connect to it and run commands.

Normally, we use the `-it` options to create a container. This is what we used a few examples ago. The `-i` option triggers interactive mode, while the `-t` option gives us a `pseudo-TTY`. At the end of the command, we tell the container to run the Bash shell. The `-d` option runs the container in the background.

It may seem relatively useless to have another Bash shell running in the background that isn't actually performing a task. But these are just simple examples to help you get the hang of Docker. A more common use case may be to run a specific application. In fact, you can even serve a website from a Docker container by installing and configuring Apache within the container, including a virtual host. The question then becomes: how do you access the container's instance of Apache within a web browser? The answer is **port redirection**, which Docker also supports. Let's give this a try.

First, let's create a new container in detached mode. Let's also redirect port `80` within the container to port `8080` on the host:

```
docker run -dit -p 8080:80 ubuntu /bin/bash
```

The command will output a container ID. This ID will be much longer than you're accustomed to seeing. This is because when we run `docker ps -a`, it only shows shortened container IDs. You don't need to use the entire container ID when you attach; you can simply use part of it as long as it's long enough to be different from other IDs:

```
docker attach dfb3e
```

Here, I've attached to a container with an ID that begins with `dfb3e`. I'm now attached to a Bash shell within the container.

Let's install Apache. We've done this before, but there are a few differences that you'll see. First, if you simply run the following command to install the `apache2` package as we would normally do, it may fail for one or two reasons:

```
sudo apt install apache2
```

The two problems here are first that `sudo` isn't included by default in the Ubuntu container, so it won't even recognize the `sudo` part of the command. When you run `docker attach`, you're actually attaching to the container as the `root` user, so the lack of `sudo` won't be an issue anyway. Second, the repository index in the container may be out of date, if it's even present at all. This means that `apt` within the container won't even find the `apache2` package. To solve this, we'll first update the repository index:

```
apt update
```

Then, install `apache2` using the following command:

```
apt install apache2
```

You may be asked to set your time zone or geographic location during the installation of packages. If so, go ahead and enter each prompt accordingly.

Now we have Apache installed in our container. We don't need to worry about configuring the default sample web page or making it look nice. We just want to verify that it works. Let's start the service:

```
/etc/init.d/apache2 start
```

Apache should be running within the container. Now, press *Ctrl* + *p* and then press *q* (don't let go of the *Ctrl* key while you press these two letters) to exit the container, but allow it to keep running in the background. You should be able to visit the sample Apache web page for the container by navigating to `localhost:8080` in your web browser. You should see the default `It works!` page of Apache.

Congratulations, you're officially running an application within a container.

As your Docker knowledge grows, you'll want to look deeper into the concept of an `ENTRYPOINT`. An `ENTRYPOINT` is a preferred way of starting applications in a Docker container. In our examples so far, we've used an `ENTRYPOINT` of `/bin/bash`. While that's perfectly valid, an `ENTRYPOINT` is generally a Bash script that is configured to run the desired application and is launched by the container.

Our Apache container is running happily in the background, responding to HTTP requests over port `8080` on the host. But what should we do with it at this point? We can create our own image from it so that we can simplify deploying it later. To be fair, we've only installed Apache inside the container, so it's not saving us that much work. In a real production environment, you may have a container running that needed quite a few commands to set up. With an image, we can have all of that work baked into the image so we won't have to run any setup commands we may have each time we want to create a container. To create a container image, let's grab the container ID of a running container by running the `docker ps` command. Once we have that, we can now create a new image of the container with the `docker commit` command:

```
docker commit <Container ID> ubuntu/apache-server:1.0
```

That command will return us the ID of our new image. To view all the Docker images available on our machine, we can run the `docker images` command to have Docker return a list. You should see the original Ubuntu image we downloaded, along with the one we just created. We'll first see a column for the repository the image came from; in our case it is Ubuntu. Next, we see the tag. Our original Ubuntu image (the one we used `docker pull` to download) has a tag of `latest`. We didn't specify that when we first downloaded it; it just defaulted to `latest`. In addition, we see an image ID for both, as well as the size.

To create a new container from our new image, we just need to use `docker run`, but specify the tag and name of our new image. Note that we may already have a container listening on port `8080`, so this command may fail if that container hasn't been stopped:

```
docker run -dit -p 8080:80 ubuntu/apache-server:1.0 /bin/bash
```

Speaking of stopping a container, I should probably show you how to do that as well. As you can probably guess, the command is `docker stop` followed by a container ID:

```
docker stop <Container ID>
```

This will send the `SIGTERM` signal to the container, followed by `SIGKILL` if it doesn't stop on its own after a delay.

Admittedly, the Apache container example was fairly simplistic, but it does the job as far as showing you a working example of a container that is actually somewhat useful. Before continuing on, think for a moment of all the use cases you can use Docker for in your organization. It may seem like a very simple concept (and it is), but it allows you to do some very powerful things. Perhaps you'll want to try to containerize your organization's intranet page, or some sort of application. The concept of Docker sure is simple, but it can go a long way with the right imagination.

Before I close out this section, I'll give you a personal example of how I implemented a container at a previous job. At this organization, I worked with some Embedded Linux software engineers who each had their own personal favorite Linux distribution. Some preferred Ubuntu, others preferred Debian, and a few even ran Gentoo. This in and of itself wasn't necessarily an issue—sometimes it's fun to try out other distributions. But for developers, a platform change can introduce inconsistency, and that's not good for a software project. The build tools are different in each distribution, because they all ship different versions of all development packages and libraries. The application this particular organization developed was only known to compile properly in Debian, and newer versions of the compiler posed a problem for the application. My solution was to provide each developer with a Docker container based on Debian, with all the build tools that they needed to perform their job baked in. At this point, it no longer mattered which distribution they ran on their workstations. The container was the same no matter what they were running. Regardless of what their underlying operating system was, they all had the same tools. This gave each developer the freedom to run their preferred distribution of Linux (or even macOS) and it didn't impact their ability to do their job. I'm sure there are some clever use cases you can come up with for implementing containerization.

Now that we understand the basics of Docker, let's take a look at automating the process of building containers.

## Automating Docker image creation with Dockerfiles

I've mentioned previously in this book that anything worth having a server do more than once should be automated, and building a Docker container is no exception. A Dockerfile is a neat way of automating the building of Docker images by creating a text file with a set of instructions for their creation. Docker is able to take this file, execute the commands it contains, and build a container. It's magic.

The easiest way to set up a `Dockerfile` is to create a directory, preferably with a descriptive name for the image you'd like to create (you can name it whatever you wish, though), and inside it create a text file named `Dockerfile`. For a quick example, copy this text into your `Dockerfile` and I'll explain how it works:

```
FROM ubuntu
MAINTAINER Jay <jay@somewhere.net>

# Avoid confirmation messages
ARG DEBIAN_FRONTEND=noninteractive

# Update the container's packages
RUN apt update; apt dist-upgrade -y

# Install apache2 and vim
RUN apt install -y apache2 vim-nox

# Start Apache
ENTRYPOINT apache2ctl -D FOREGROUND
```

Let's go through this Dockerfile line by line to get a better understanding of what it's doing:

```
FROM ubuntu
```

We need an image to base our new image on, so we're using Ubuntu as a starting point.

This will cause Docker to download the `ubuntu:latest` image from Docker Hub, if we haven't already downloaded it locally. If we do have it locally, it will just use the locally cached version.

```
MAINTAINER Jay <myemail@somewhere.net>
```

Here, we're setting the maintainer of the image. Basically, we're declaring its author. This is optional, so you don't need to include that if you don't want to.

```
# Avoid confirmation messages
```

Lines beginning with a hash symbol (#) are ignored, so we are able to create comments within the Dockerfile. This is recommended to give others a good idea of what your Dockerfile is doing.

```
ARG DEBIAN_FRONTEND=noninteractive
```

Here, we're setting an environment variable that in this case sets the environment to `noninteractive`. The reason we do this is that the process of building a Docker container should be automatic; if a prompt comes up and asks you a question, your input will not pass through and the process will hang. With this environment variable, we're clarifying that we want to be in `noninteractive` mode so that the default answers to any questions that come up will be used and we won't be prompted.

```
RUN apt update; apt dist-upgrade -y
```

With the `RUN` command, we're telling Docker to run a specific command while the image is being created. In this case, we're updating the image's repository index and performing a full package update to ensure the resulting image is as fresh as can be. The `-y` option is provided to suppress any requests for confirmation while installing packages. Despite the fact that we set `noninteractive` mode earlier, `apt` will still try to confirm changes interactively, and the `-y` option suppresses that.

```
RUN apt install -y apache2 vim-nox
```

Next, we're installing both `apache2` and `vim-nox`. The `vim-nox` package isn't required, but I personally like to make sure all of my servers and containers have it installed. I mainly included it here to show you that you can install multiple packages in one line.

```
ENTRYPOINT apache2ctl -D FOREGROUND
```

I mentioned the concept of an `ENTRYPOINT` earlier, which again is where we clarify which application should run when the container starts. The `apache2ctl` command is a wrapper command for Apache that allows administrators to control the finer points of running the Apache daemon. A full walkthrough of this command is beyond the scope of this chapter, but we're using it here because we want Apache to automatically start with the container, and `apache2ctl` is one method of doing that without relying on `systemctl` (which the container doesn't have).

Great, now we have a Dockerfile. So, what do we do with it? Well, turn it into an image of course! To do so, we can use the `docker build` command, which can be executed from within the directory that contains the Dockerfile. Here's an example of using the `docker build` command to create an image tagged `packt/apache-server:1.0`:

```
docker build -t packt/apache-server:1.0 .
```

Once you run that command, you'll see Docker create the image for you, running each of the commands you asked it to. The image will be set up just the way you like. Basically, we just automated the entire creation of the Apache container we used as an example in this section. If anything goes wrong, Docker will print an error to your shell. You can then fix the error in your Dockerfile and run it again, and it will continue where it left off.

Once complete, we can create a container from our new image:

```
docker run -dit -p 8080:80 packt/apache-server:1.0
```

Almost immediately after running the container, the sample Apache site will be available on `localhost:8080` on the host. With a Dockerfile, you'll be able to automate the creation of your Docker images. That was easy, wasn't it? There's much more you can do with Dockerfiles; feel free to peruse Docker's official documentation to learn more. Exploration is key, so give it a try and experiment with it.

## Managing LXD containers

With Docker out of the way, let's take a look at how to run containers with LXD. Let's dive right in and install the required package:

```
sudo snap install lxd
```

As you can see, installing LXD is just as easy as installing Docker. In fact, managing containers with LXD is very straightforward as well, as you'll soon see. Installing LXD gives us the `lxc` command, which is the command we'll use to manage LXD containers. Before we get going though, we should add our user account to the `lxd` group:

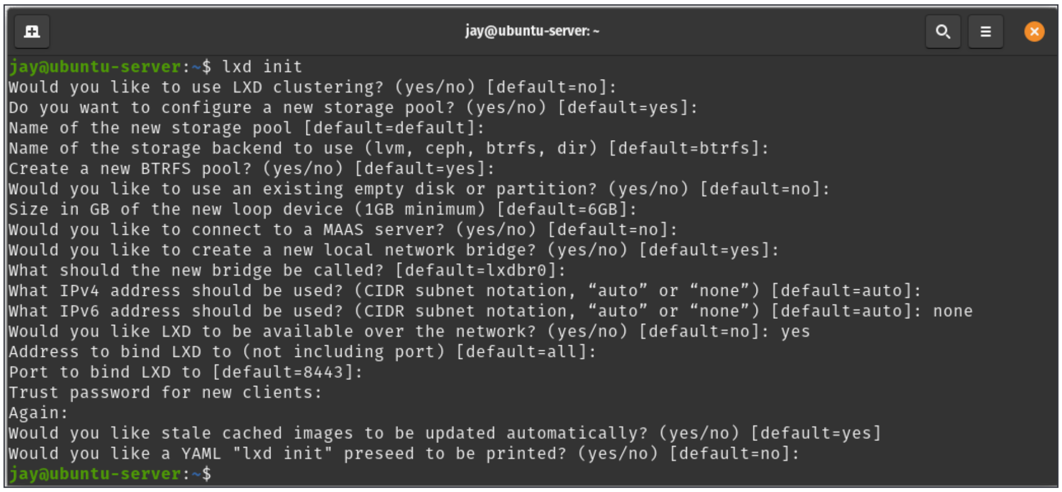
```
sudo usermod -aG lxd <yourusername>
```

Make sure you log out and log in for the changes to take effect. Just like with the `docker` group with Docker, the `lxd` group will allow our user account to manage LXD containers.

Next, we need to initialize our new LXD installation. We'll do that with the `lxd init` command:

```
lxd init
```

The output will look similar to the following screenshot:



```
jay@ubuntu-server:~$ lxd init
Would you like to use LXD clustering? (yes/no) [default=no]:
Do you want to configure a new storage pool? (yes/no) [default=yes]:
Name of the new storage pool [default=default]:
Name of the storage backend to use (lvm, ceph, btrfs, dir) [default=btrfs]:
Create a new BTRFS pool? (yes/no) [default=yes]:
Would you like to use an existing empty disk or partition? (yes/no) [default=no]:
Size in GB of the new loop device (1GB minimum) [default=6GB]:
Would you like to connect to a MAAS server? (yes/no) [default=no]:
Would you like to create a new local network bridge? (yes/no) [default=yes]:
What should the new bridge be called? [default=lxdbr0]:
What IPv4 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]:
What IPv6 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]: none
Would you like LXD to be available over the network? (yes/no) [default=no]: yes
Address to bind LXD to (not including port) [default=all]:
Port to bind LXD to [default=8443]:
Trust password for new clients:
Again:
Would you like stale cached images to be updated automatically? (yes/no) [default=yes]
Would you like a YAML "lxd init" preseed to be printed? (yes/no) [default=no]:
jay@ubuntu-server:~$
```

Figure 17.3: Setting up LXD with the lxd init command

The `lxd init` command will ask us a series of questions regarding how we'd like to set up LXD. The defaults are mostly fine for everything, and for the size of the pool, I just used 6 GB but you can use whatever size you want to. I set `ipv6` to `none` during the setup since my network doesn't utilize that, and I also decided to make `lxd` available over the network.

Even though we chose the defaults for most of the questions, they'll give you a general consensus of some of the different options that LXD makes available for us. For example, we can see that LXD supports the concept of a storage pool, which is one of its neater features. Here, we're creating a default storage pool with a filesystem format of `btrfs`, which is a filesystem that is used on actual hard disks. In fact, we could even use ZFS if we wanted to, which just goes to show you how powerful this technology is. During the setup process, LXD sets up the storage pool, network bridge, IP address scheme, and basically everything we need to get started.

Now that LXD is installed and set up, we can configure our first container:

```
lxc launch ubuntu:20.04 mycontainer
```

With that simple command, LXD will now download the root filesystem for this container and set it up for us. Once done, we'll have an LXD container based on Ubuntu 20.04 running and available for use. This is different than Docker, which only sets up an image by default, making us run it manually. During this process, we gave the container a name of `mycontainer`. The process should be fairly easy to follow so far.

You might be wondering why we used a `lxc` command to create a container, since we're learning about LXD here. As I mentioned earlier, LXD is an improvement over LXC, and as such it uses `lxc` commands for management. Commands that are specific to the LXD layer will be `lxd`, and anything specific to container management will be done with `lxc`.

When it comes to managing containers, there are several types of operations you will want to perform, such as listing containers, starting a container, stopping a container, deleting a container, and so on. The `lxc` command suite is very easy and straightforward. Here is a table listing some of the most common commands you can use, and I'm sure you'll agree that the command syntax is very logical. For each example, you substitute `<container>` with the name of the container you created:

| **Goal** | **Command** |
|---|---|
| List containers | `lxc list` |
| Start a container | `lxc start <container>` |
| Stop a container | `lxc stop <container>` |
| Remove a container | `lxc delete <container>` |
| List downloaded images | `lxc image list` |
| Remove an image | `lxc image delete <image_name>` |

With all the basics out of the way, let's jump into our container and play around with it. To open a shell to the container we just created, we would run the following:

```
lxc exec mycontainer bash
```

In the preceding command, `exec` tells the container we want to execute a command, `mycontainer` is the name of the container that we want to execute something against, and the specific command we want to execute is `bash`. After you execute that command, it immediately runs `bash` from the container as `root`. From here, you can configure the container as you need to, installing packages, setting up services, or whatever else you may need to do in order to make the container conform to the purpose you have for it. In fact, the process of customizing the container for redeployment is actually easier than it is with Docker. Unlike with Docker, changes are not wiped when you exit a container, and you don't have to exit it a particular way to avoid losing your changes. We also don't have layers to deal with in LXD, which you may or may not be happy about (layers in Docker containers can make deployments faster, but when previously run containers aren't cleaned up, it can look messy).

The Ubuntu image we used to create our container includes a default user account, `ubuntu`. This is similar to some VPS providers, which also include an `ubuntu` user account by default (Amazon EC2 is an example of this). If you prefer to log in as this user rather than `root`, you can do that with this command:

```
lxc exec mycontainer -- su --login ubuntu
```

The `ubuntu` user has access to `sudo`, so you'll be able to run privileged tasks with no issue.

To exit the container, you can press *Ctrl* + *d* on your keyboard, or simply type `exit`. Feel free to log in to the container and make some changes and experiment. Once you have the container set up the way you like it, you may want the container to automatically start up when you boot your server. This is actually very easy to do:

```
lxc config set mycontainer boot.autostart 1
```

With the preceding command, we're setting `boot.autostart` to `1`, which turns on that particular feature. Similar to a Boolean variable for those that are familiar with programming, `1` means "on" and `0` means "off." After setting this config value, your newly created container will now start up with the server anytime it's booted.

Now, let's have a bit of fun. Feel free to install the `apache2` package in your container. Similar to Docker, I've found that you will probably want to run `apt update` to update your package listings first, as I've seen failures installing packages on a fresh container solely because the indexes were stale. So, just run this to be safe:

```
sudo apt update && sudo apt install apache2
```

Now, you should have Apache installed and running in the container. Next, we need to grab the IP address of the container. Yes, you read that right, LXD has its own IP address space for its containers, which is very neat. Simply run `ip addr show` (the same command you'd run in a normal server) and it will display the IP address information. On the same machine that's running the container, you can visit this IP address to see the default Apache web page. If you're running the container on a server with no graphical user interface, you can use the `curl` command to verify that it's working:

```
curl <container_ip_address>
```

Although we have Apache running in our container, we can see that it's not very useful yet. The web page is only available from the machine that's hosting the container. This doesn't help us much if we want users in our local network or even from the outside internet to be able to reach our site. We could set up firewall rules to route traffic to it, but there's an easier way—creating a profile for external access. I mentioned earlier that even though LXD is a containerization technology, it shares some of its feature set with VMs, basically giving you VM-like features in a non-VM environment. With LXD, we can create a profile to allow it to get an IP address from your DHCP server and route traffic directly through your LAN, just as with a physical device that you connect to your network.

Before continuing, you'll need a bridge connection set up on your server. This is done in software via Netplan and was discussed as part of the previous chapter. If you list your network interfaces (`ip addr show`), you should see a `br0` connection. If you don't have this configured, refer back to *Chapter 16*, *Virtualization*, and refer to the *Bridging the virtual machine network* section there. Once you've created this connection, you can continue on.

> Some network cards do not support bridging, especially with some Wi-Fi cards. If you're unable to create a bridge on your hardware, the following section may not work for you. Consult the documentation for your hardware to ensure your network card supports bridging.

To create the profile we'll need in order to enable external access to our containers, we'll use the following command:

```
lxc profile create external
```

We should see output similar to the following:

```
Profile external created
```

Next, we'll need to edit the profile we just created. The following command will open the profile in a text editor so that you can edit it:

```
lxc profile edit external
```

Inside the profile, we'll replace its text with this:

```
description: External access profile
devices:
  eth0:
  name: eth0
```

```
    nictype: bridged
    parent: br0
    type: nic
```

From this point forward, we can launch new containers with this profile with the following command:

```
lxc launch ubuntu:20.04 mynewcontainer -p default -p external
```

Notice how we're applying two profiles, `default` and then `external`. We do this so the values in `default` can be loaded first, followed by the second profile so that it overrides any conflicting parameters that may be present.

We already have a container, though, so you may be curious how we can edit the existing one to take advantage of our new profile. That's simple:

```
lxc profile add mycontainer external
```

From this point forward, assuming the host bridge on your server is configured properly, the container should be accessible via your local LAN. You should be able to host a resource, such as a website, and have others be able to access it. This resource could be a local intranet site or even an internet-facing web site.

As far as getting started with LXD is concerned, that's essentially it. LXD is very simple to use and its command structure is very logical and easy to understand. With just a few simple commands, we can create a container, and even make it externally accessible. Canonical has many examples and tutorials available online to help you push your knowledge even further, but with what you've learned so far, you should have enough practical knowledge to roll out this solution in your organization.

## Summary

Containers are a wonderful method of hosting applications. Without needing to dedicate RAM and CPU resources, you can spin up more containers on your hardware than you'd be able to with VMs. While not all applications can be run inside containers, it's a very useful tool to have available. In this chapter, we looked at both Docker and LXD. While Docker is better for cross-platform applications, LXD is simpler to use but is very flexible. We started out by discussing the differences between these two solutions, then we experimented with both, creating containers and looking at how to manage them.

In the next chapter, we will expand our knowledge of containers even further and take a look at orchestration, which allows us to manage multiple containers more efficiently. This will be the chapter where all of the concepts relating to containers come together.

## Further reading

- Docker documentation page: `https://docs.docker.com/`
- LXD introduction: `https://linuxcontainers.org/lxd/introduction/`
- LXD documentation: `https://ubuntu.com/server/docs/containers-lxd`

# 18

# Container Orchestration

In the previous chapter, we started learning concepts around containerization. We've learned what containers are and how they differ from **Virtual Machines** (**VMs**), as well as how to run two different types of containers (Docker and LXD). As you are now aware, containers are typically lightweight (which means you run a larger number of them than VMs on the same hardware) and are easy to manage with a command syntax that's rather logical, such as `docker run myapp` to launch a container named `myapp`. Depending on the size of your organization, you may only need to run one or two containers to suit your needs, or perhaps you plan on scaling up to hundreds of them. While it's rather simple to maintain a small number of containers, the footprint can quickly expand and become much harder to keep track of.

In this chapter, we'll start looking into the concept of **Container Orchestration**, which can help us to better maintain the containers that we run. Without such orchestration, the onus is on you, the administrator, to ensure your containers are running properly. While someone still needs to be responsible for mission-critical containers, orchestration provides us with tools we can use to manage them much more efficiently. In addition, orchestration allows us to create an entire cluster that's easily scalable, so we're better equipped to handle the demand of our users or clients. As we navigate this topic, we will work on:

- Container orchestration
- Preparing a lab environment for Kubernetes testing
- Utilizing MicroK8s
- Setting up a Kubernetes cluster
- Deploying containers via Kubernetes

Excited? I know I am—containerization can be a lot of fun to learn and work with. We'll even create our very own cluster in this chapter! But before we can do that, let's make sure we have an understanding of what container orchestration is.

## Container orchestration

In the last chapter, we covered the basics of running containers on your server. Of special note is the coverage of Docker, which will play a very important role in this chapter. We saw how to pull a Docker image as well as how to use such an image to create a container. There are many more advanced concepts we can learn when it comes to Docker, but understanding the essentials is good enough for the scope of this chapter. And now that we know how to run containers, looking further into how to more efficiently manage them is a logical next step.

Traditionally, as an administrator, you'll ensure the critical apps and services for your organization are always healthy and available. If a critical resource stops working for any reason, it falls on you to return it to a healthy state. Regardless of whether we're utilizing applications on a physical server, or in a VM or container, this need doesn't change—production apps need to be available at all times with minimal or no downtime. Specific to containers, what orchestration does for us is help us maintain them much more efficiently. Orchestration allows us to not only manage our containers in one place, but it also provides us with additional tooling we can use to more intelligently handle load and recover from faults.

Consider this example: let's assume you work at an organization that has an important website that needs to be available online at all times, and it's currently set up inside a VM. As the administrator, you test the application by running it inside a Docker container and discover that it not only functions the same as in the VM, it also requires less of the server's memory to run it in a container, and it's even more responsive than before. Everyone at your company is impressed, and the project of converting your company's site to run inside a container is a complete success.

Now, let's assume your organization is getting ready to release a brand new version of your company's product. It's expected that the demand for your website will increase ten times for at least the first few weeks after the debut. To address this, you can launch however many additional containers you feel will handle the expected load, and then set up a load balancer to spread traffic evenly between them. When the excitement over the new release winds down, you can remove the newly added containers to return to your original population. We haven't gone over load balancers yet, but these are features built into Docker and you don't need to install anything extra to take advantage of this.

If the launch of your organization's newly updated product is expected to go live at 1:00am, you'd have to make sure you're awake then and execute the necessary commands to launch the extra containers and deploy the load balancer. You would then watch the containers for a while and ensure they're stable. Perhaps you've already tested this in a testing environment, so you have reasonable confidence that the maintenance will go smoothly. After you successfully launch the containers, you set a reminder for yourself in your calendar to log back in after two weeks to undo the changes.

While that approach can technically work and result in a successful rollout, it's not very efficient. Sure, you *can* log in at 1:00am to launch the extra containers and deploy the load balancer, but is that an effective use of your time? What if you're very sleepy at that time and make a mistake that results in a failure during an important launch? And when the launch window is over and the load returns to normal, you may or may not see the alert that you've intended to use to remind yourself and lower the container count. In that situation, you could end up with an expensive bill since your server would be using extra energy to run containers that were no longer needed. Even worse, manually managing your containers means that it can't handle a situation where the load increases unexpectedly. Despite our best intentions, any process that is run manually may or may not work out well; we're only human after all.

With container orchestration, we essentially delegate the process of running containers to an application that will automatically create additional containers as demand increases and remove unneeded containers when demand winds down. Orchestration also simplifies the process of setting up applications, by giving us tools we can use to ensure a particular number of containers are running at a minimum and we can set a maximum for when load and demand spikes. This empowers us to have infrastructure that grows and shrinks automatically to match the needs of our users. In addition, container orchestration allows us to automatically heal from failures. If a container runs into a problem and fails for some reason, it will be deleted and recreated from the image.

As you can see, orchestration gives us additional value, it can actually save you time and effort. To be fair, some of the features mentioned are part of containerization itself and not necessarily specific to orchestration, but the latter does give us the ability to manage these tools much more efficiently. Is container orchestration for everyone? No technology meets 100% of all use-cases, but it's certainly something to consider. If you only run a single container and have no plans to run another, then a technology such as orchestration is overkill—it would take you more time to manage the cluster than it would to manage the container itself. Use your best judgment.

If you do decide to implement containerization, **Kubernetes** (often abbreviated **K8s**) is a very popular solution for container orchestration and is what we'll be exploring in this chapter. What Kubernetes allows us to do is create deployments for our applications, providing us with fine-tuned control we can leverage to determine the actual behavior of our applications. It can automatically re-spawn a container if it stops working for any reason, ensure the number of containers running meets the demand, and spread our workloads across many worker nodes so that no one server becomes overwhelmed with running too many containers. Kubernetes is very popular and is not specific to Ubuntu Server. However, utilizing a technology such as Kubernetes in Ubuntu is one of many ways that we can use to extend the platform.

What do you need in order to set up a Kubernetes cluster? That's likely a logical question to have at this point. In the next section, we'll cover some of the considerations to make before deciding how to set it up.

## Preparing a lab environment for Kubernetes testing

In an organization, planning a roll-out of an entire Kubernetes cluster can be fairly involved—you may have to purchase hardware and also analyze your existing environment and understand how containerization will fit in. It's possible that some applications you want to run aren't a good fit for containers; some don't support running in a container at all. Assuming you've already checked the documentation for the applications you are wanting to run in containers and came to the conclusion that such a technology is supported, the next step is procuring the hardware (if you don't already have a place to run it) and then setting up the cluster.

Specific to us in this book, we don't need to contact a server vendor and submit a purchase order to simply test out the technology. If you are actually involved with the rollout of container orchestration at your organization, then it's a fun project to work on. But for the purposes of this book, let's discuss some of the details around what's needed to set up a personal lab in order to set up Kubernetes and start learning the platform.

The best rule of thumb when setting up a lab for testing software is to try to use what you have available. To create a Kubernetes cluster, we'll need one Ubuntu machine to act as the master, and one or more additional servers to be used as worker nodes. As you'll learn later in the book, the goal of Kubernetes is to schedule containers to run in **Pods** on worker nodes, and for that it's more effective to have more than just one worker to go along with your master.

But on what type of device should you run Kubernetes on in order to test it out and learn it? For this, we have the same possibilities as we discussed when installing Ubuntu Server back in *Chapter 1*, *Deploying Ubuntu Server*—we can use VMs, physical servers, spare desktops or laptops, as well as Raspberry Pis (or any combination of those). Again, use whatever you have available. For a rollout in an organization, you may end up using a virtualization server for the master and worker nodes, or perhaps physical servers. One of my favorite platforms for Kubernetes is the Raspberry Pi, believe it or not. I've been running a Kubernetes cluster in production for quite some time that consists of only Raspberry Pi 4 units with complete success. If nothing else, purchasing a few Pis is a relatively low barrier to entry. You can even utilize a cloud provider if you'd like, though doing so goes beyond the scope of this chapter as cloud providers generally have their own tools in place to manage clusters.

In general, it's recommended to have a master node and a handful of workers. A single worker will suffice, but in this chapter, I'll show the process of setting up two workers to better understand how the platform scales. On your end, you can use a single master and worker, or set up however many workers you'd like. One of the best things about Kubernetes is that you aren't stuck with the number of workers you set up initially, you can add more nodes to your cluster as time goes on. An organization may start with just a few workers but add additional ones as the need arises.

As far as resources go, requirements for Kubernetes are relatively modest. The node that's designated as the master should have a minimum of two CPUs or one CPU with at least two cores. Obviously, it's better if you have more than that, such as a quad-core CPU for the master, but two cores are the minimum. The worker nodes can have a single CPU core each, but the more cores they have, the more Pods they'll be able to run. When it comes to RAM, I recommend a minimum of 2 GB of memory on each, but again, if you have more than that on each node then that's even better.

> It's important that the IP addresses for the nodes in your cluster do not change. If an IP of a cluster node changes, it can cause the cluster nodes to be unable to find each other. In *Chapter 10*, *Connecting to Networks*, we learned how to set up a static IP address, which is a good solution for preventing IP addresses from changing. As an alternative, you can set up a static lease in your DHCP server if you wish. It doesn't matter which solution you use, so long as you prevent the IP addresses of cluster nodes from changing.

For some readers, a more accessible method of setting up comes in the form of **MicroK8s**, which allows you to even set up Kubernetes on a single machine. If your only goal is to set up a simple test installation, it's one of the best and easiest methods of getting started. MicroK8s is not recommended for running a production cluster in an organization, but it's definitely a great way to learn. I do recommend that you work through the standard Kubernetes procedure with multiple nodes if you can, but in the next section we'll walk through setting up MicroK8s for those of you that would like to utilize that method.

## Utilizing MicroK8s

If you don't have more than one machine or enough memory on your laptop or desktop to run multiple nodes inside virtualization software such as VirtualBox, MicroK8s is a simple way to set up a Kubernetes instance for testing the platform, as well as going through the examples in this chapter. MicroK8s is actually provided by Canonical, the makers of Ubuntu. That just goes to show you how important Kubernetes is to the Ubuntu distribution, its own creator is going the extra mile to contribute to the platform. MicroK8s is available for Linux, macOS, as well as Windows. So regardless of which operating system you're running on your laptop or desktop, you should be able to install and use it. If nothing else, it gives you a great test installation of Kubernetes that will come in handy as you learn.

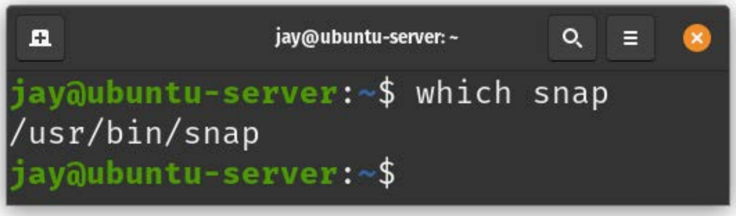
To set it up, follow along with one of the subsections below that matches the operating system installed on your computer.

## Installing MicroK8s on Linux

When it comes to Linux, MicroK8s is distributed as a snap package. We covered snap packages back in *Chapter 3*, *Managing Software Packages*, and it's a great solution for cross-distribution package management. If you run a recent version of Ubuntu on your computer, then you should already have support for snap packages and you can proceed to install MicroK8s. If you're running a distribution of Linux on your computer other than Ubuntu, then you may not have access to the `snap` command by default. If in doubt, you can use the `which` to see if the command is available:

```
which snap
```

If you do have the ability to install snap packages, then your output should be similar to the following:



```
jay@ubuntu-server:~$ which snap
/usr/bin/snap
jay@ubuntu-server:~$
```

Figure 18.1: Checking if the snap command is available

If you see no output when you run `which snap`, then that means your distribution doesn't have support for this package type installed. Canonical makes the `snap` command available to distributions outside of Ubuntu, so if you're using a distribution other than Ubuntu, it's usually just a matter of installing the required package. Canonical has additional information available on the following site, which shows the process of enabling `snap` for several distributions: `https://snapcraft.io/docs/installing-snapd`.

For example, the documentation on that site gives the following command to install the required package for Fedora:

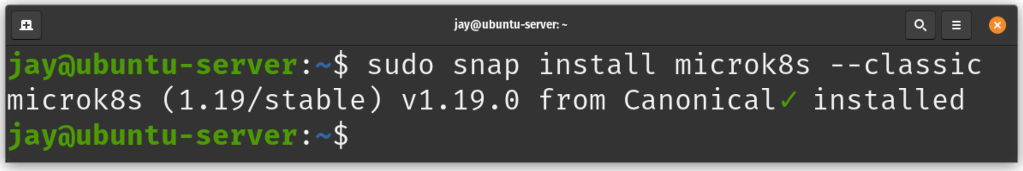
```
sudo dnf install snapd
```

Essentially, as long as you follow along with the instructions for setting up `snap` that are specific to the distribution of Linux you're running, the process should be simple enough. If for some reason your distribution isn't supported, you can simply use the same Ubuntu installation as you've been using during this book, which will have `snap` support built in.

Once you either confirm you already have support for `snap` on your computer, or you successfully enable the feature, you can install MicroK8s with the following command:

```
sudo snap install microk8s --classic
```

If successful, you should see a confirmation in your terminal that the process was successful:



```
jay@ubuntu-server:~$ sudo snap install microk8s --classic
microk8s (1.19/stable) v1.19.0 from Canonical✓ installed
jay@ubuntu-server:~$
```

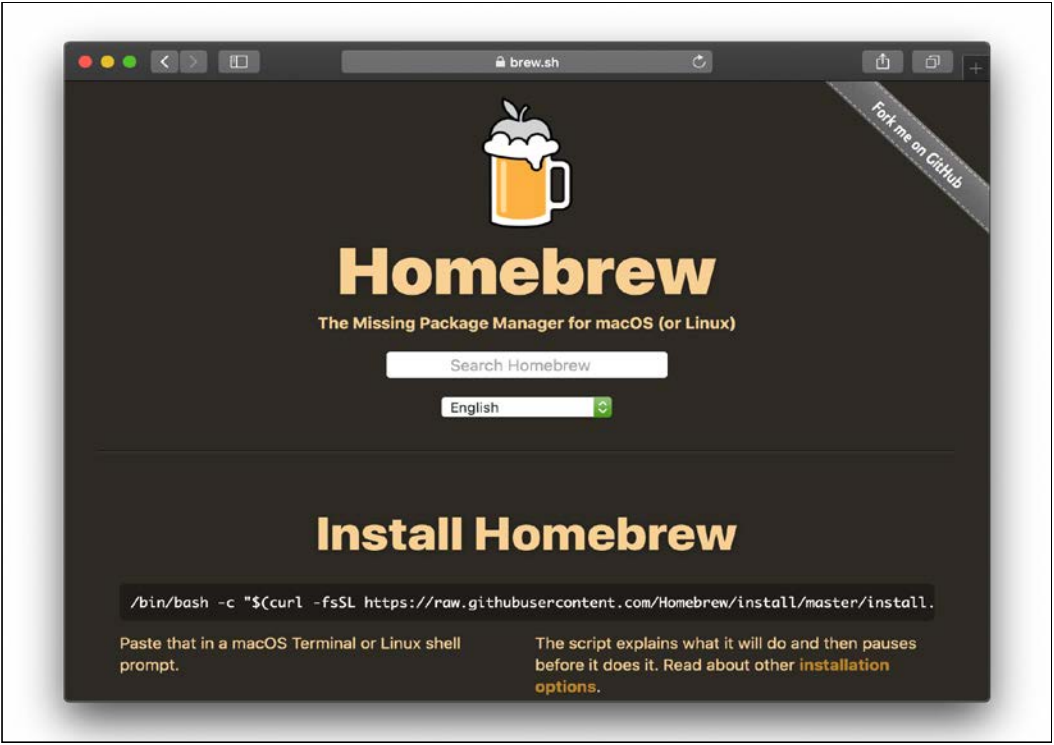
Figure 18.2: Setting up MicroK8s on a Linux installation

Now that we have MicroK8s installed on your Linux computer, we can proceed through the chapter. Or, you can check out the next section to see the installation process for macOS.

## Installing MicroK8s on macOS

On macOS, the installation process for MicroK8s is similar. The process for Mac will utilize **Homebrew**, which is an addon for macOS. Homebrew isn't specific to Kubernetes or MicroK8s; it gives you the ability to install additional packages on your Mac that aren't normally available, with MicroK8s being one of many. Homebrew is essentially a command-line utility with syntax similar to Ubuntu's `apt` command.

To install Homebrew, visit `https://brew.sh` in your browser, and the command required to set it up will be right there on the site. I could insert the command to install Homebrew right here on this page, but the process may change someday, so it's better to get the command right from the official website for the utility. At the time of writing, the page looks like this:



```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.
```

Paste that in a macOS Terminal or Linux shell prompt.

The script explains what it will do and then pauses before it does it. Read about other installation options.

Figure 18.3: The Homebrew website

In the screenshot, you'll notice the command with white text on a black highlight. It's cut off a bit, due to the command being wider than this page. You should see the entire command on the site, so you can copy it from there and paste it into your Mac's terminal.

It's becoming increasingly popular for application developers to provide a command on their website to install their application that you paste directly into your terminal. This is very convenient and allows you to set up an app quickly. But you should always research and inspect such commands before pasting them into your terminal (regardless of your operating system). It's possible that an outside actor could hijack the command on a website to trick you into running something nefarious. You can use the `wget` command to download the script their command will end up running, so you can check it to ensure it's not doing something evil. Since this method of deploying software is getting more and more common, I recommend getting into the habit of checking commands before running them (especially if they use `sudo`).

Once you have Homebrew installed on your Mac, you can proceed to install MicroK8s with Homebrew with the following command:

```
brew install ubuntu/microk8s/microk8s
```

Once that command finishes, you should have MicroK8s installed on your Mac. In the next sub-section, we'll see the same process in Windows.

## Installing MicroK8s on Windows

If you'd like to try out MicroK8s on a laptop or desktop computer running Windows 10, there's a specific installer available that will allow you to get it up and running. On the MicroK8s website, at `https://microk8s.io`, you should see a large green button labeled **Download MicroK8s for Windows**, and if you click on it, you'll be able to download the installer:

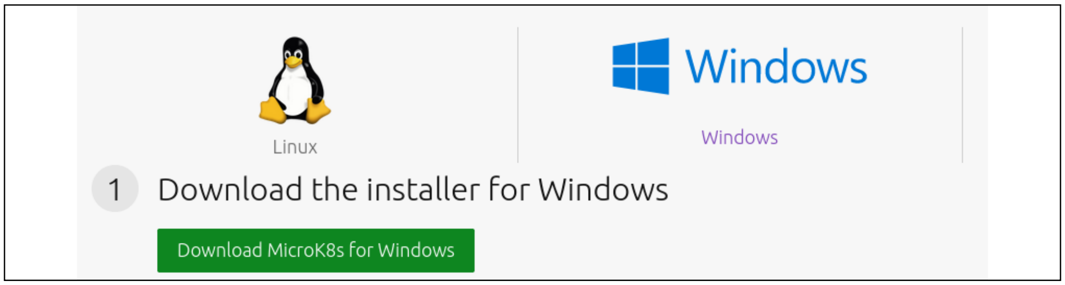
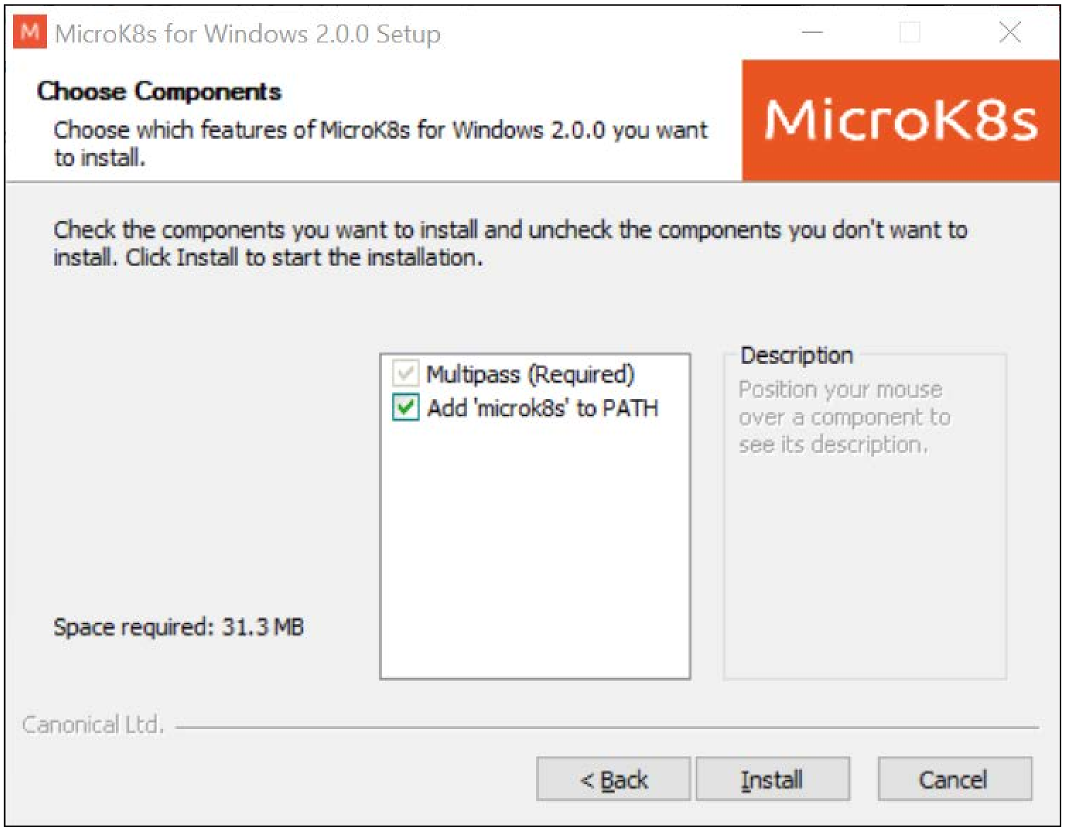

Linux
Windows
Windows
1
Download the installer for Windows
Download MicroK8s for Windows

Figure 18.4: The MicroK8s website, with a button to download the installer for Windows

Once the download finishes, you can then launch the installer, which will have several different sections, all of which you can accept the defaults for. Click **Next** and then the **Install** button to begin the actual installation:


MicroK8s for Windows 2.0.0 Setup

**Choose Components**

Choose which features of MicroK8s for Windows 2.0.0 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Install to start the installation.

- [✓] Multipass (Required)
- [✓] Add 'microk8s' to PATH

Description

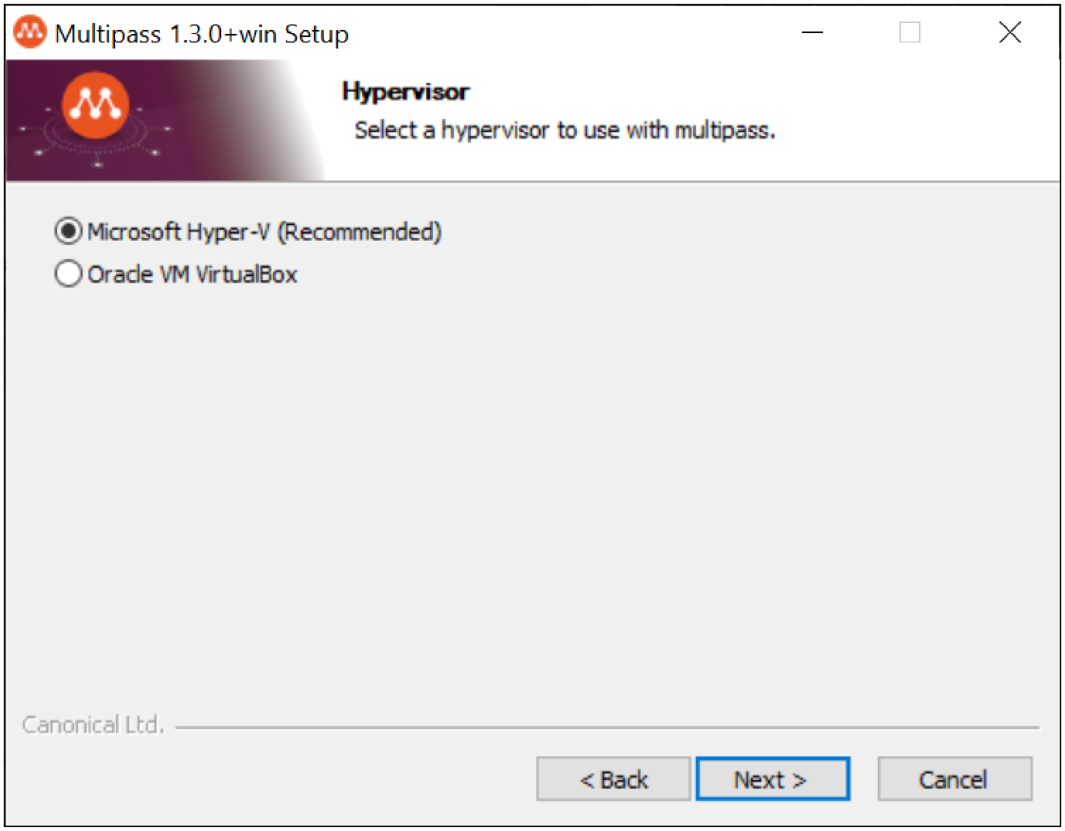
Position your mouse over a component to see its description.

Space required: 31.3 MB

Canonical Ltd.

Figure 18.5: The MicroK8s installer for Windows

After the installation begins, another installation will also launch in parallel that will ask you to select a hypervisor:


Multipass 1.3.0+win Setup

**Hypervisor**

Select a hypervisor to use with multipass.

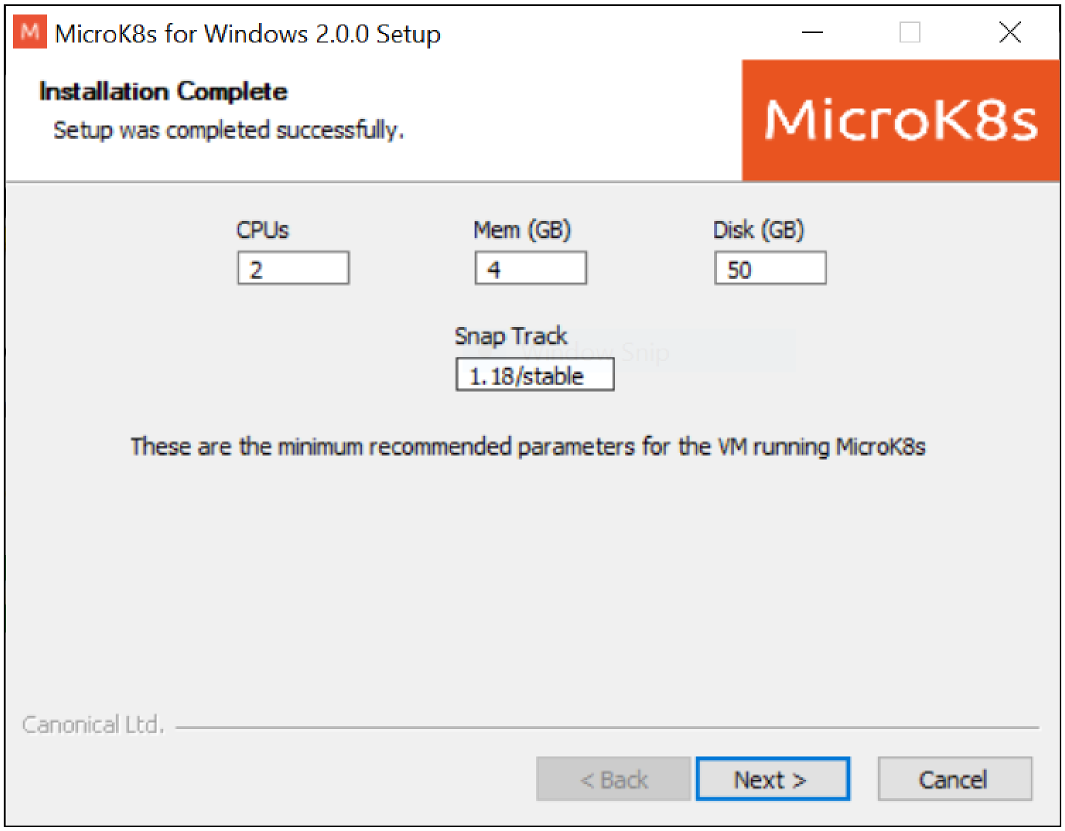
[✓ Microsoft Hyper-V (Recommended)]

Oracle VM VirtualBox

Figure 18.6: The MicroK8s installer for Windows, selecting a hypervisor

For this, you can leave the default selection on Microsoft Hyper-V, and click **Next**. You can keep the remaining options at their defaults and click **Next** through any remaining prompt that comes up. Depending on the setup of your computer, it may require you to reboot in order to finalize the required components before the installation can be completely finished. If you do see a prompt to reboot, do so, and then launch the installer again.

You'll see the following window appear, asking you to set up the parameters for MicroK8s, specifically how many virtual CPUs to utilize, how much RAM to provide, disk space, and so on:


MicroK8s for Windows 2.0.0 Setup

Installation Complete

Setup was completed successfully.

CPUs: 2

Mem (GB): 4

Disk (GB): 50

Snap Track: 1.18/stable

These are the minimum recommended parameters for the VM running MicroK8s

Canonical Ltd.

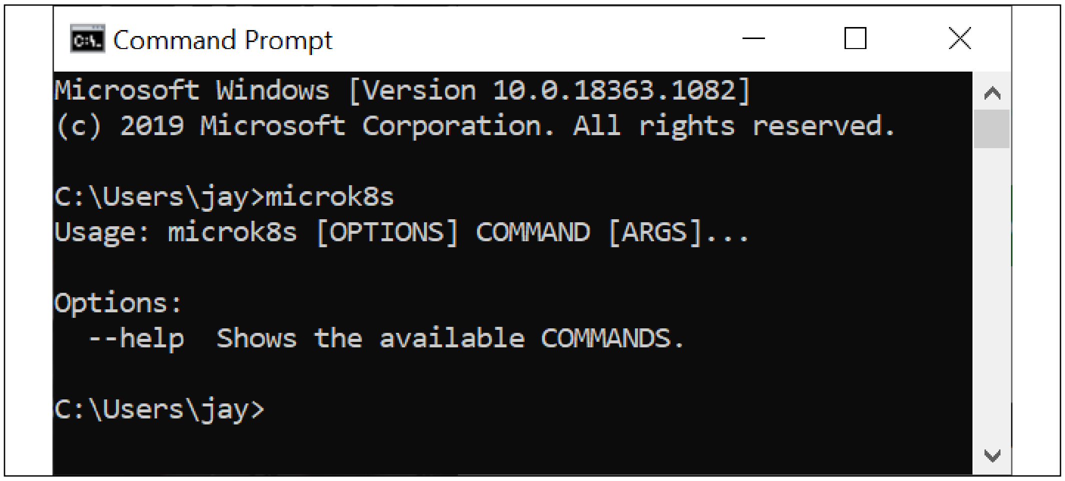
Figure 18.7: The MicroK8s installer for Windows, allocating resources

You can simply choose the defaults for this screen as well and click **Next**. Once the process is complete, you can click the **Finish** button to exit the installer.

## Interacting with MicroK8s

At this point, if you've decided to utilize MicroK8s, you should have it installed on your computer. How you actually interact with it depends on your operating system. In each case, the `microk8s` command (which we'll cover shortly) is used to control MicroK8s. Which app you use in order to enter `microk8s` commands differs from one platform to another.

On Windows, you can use the Command Prompt app, which is built into the operating system. The `microk8s` command is recognized and available for use after you install MicroK8s. When you enter the `microk8s` command by itself, it should display basic usage information:



Command Prompt

```
Microsoft Windows [Version 10.0.18363.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\jay>microk8s
Usage: microk8s [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Shows the available COMMANDS.

C:\Users\jay>
```

Figure 18.8: Executing the microk8s command with no options in a Windows Command Prompt

With macOS, you can use the built-in terminal app that's provided with the operating system. You may have another step to complete though before you can actually use MicroK8s. If you enter the `microk8s` command, and you receive an error informing you that `support for multipass needs to be set-up`, then you can run the following command to fix it:

```
microk8s install
```

If you're curious, **Multipass** is a technology that's also created by Canonical that allows you to quickly set up an Ubuntu instance for testing. It's not specific to MicroK8s or even Kubernetes, but in our case it's used in the background to facilitate MicroK8s. Multipass is not covered in this book, but it's worth taking a look at if you think you'll benefit from the ability to quickly set up Ubuntu instances to test applications and configurations on. It's available for all of the leading operating systems.

When it comes to Linux, if you've already gone through the process of setting up MicroK8s, you should be ready to use it immediately. Open up your distribution's terminal app and try the `microk8s` command to see if it's recognized. If it is, you're ready to move on.

To check the status of MicroK8s, the following command can be used to give you an overview of the various components. Before you run it, note the inclusion of `sudo` at the beginning. By default, this is required if your underlying operating system is Linux. If you're running Windows 10 or macOS, you shouldn't need `sudo`. This difference has to do with how the underlying operating system handles permissions, which is different from operating system to operating system. So, if you're not using Linux on your PC, feel free to omit `sudo` from `microk8s` commands:

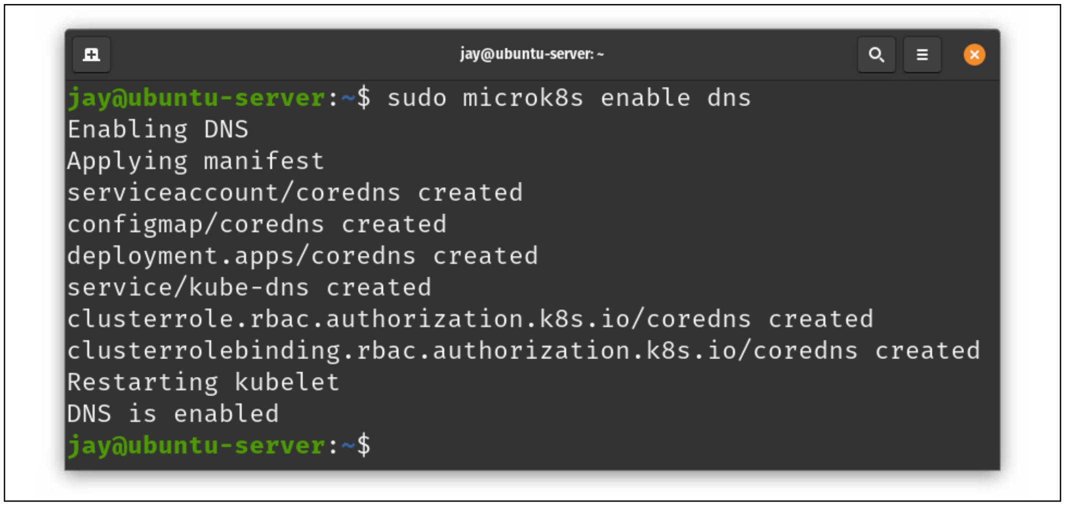
```
sudo microk8s kubectl get all --all-namespaces
```

Don't worry about the details regarding the individual components of Kubernetes at the moment, we'll cover what you need to know as we go through the rest of the chapter. For now, so long as you don't see errors when you run the previous command and check the status, you should be in good shape to continue.

By default, our MicroK8s installation comes with only the addons that are required for it to run. There are other addons we can enable, such as `storage`, which gives us the ability to expose a path on the host (the underlying operating system) to Kubernetes; `gpu`, which allows us to utilize NVIDIA GPUs within the containers; and others. We don't need to worry about these for now, but we should at least enable the `dns` addon, which sets up DNS within our cluster. It's not necessarily required, but not having it can create issues down the road when it comes to name resolution, so we may as well enable it now:

```
sudo microk8s enable dns
```

You should see output similar to the following:



```
jay@ubuntu-server:~$ sudo microk8s enable dns
Enabling DNS
Applying manifest
serviceaccount/coredns created
configmap/coredns created
deployment.apps/coredns created
service/kube-dns created
clusterrole.rbac.authorization.k8s.io/coredns created
clusterrolebinding.rbac.authorization.k8s.io/coredns created
Restarting kubelet
DNS is enabled
jay@ubuntu-server:~$
```

Figure 18.9: Enabling the DNS addon in MicroK8s on a Linux installation

In my tests, I find that the command to enable the addon seems to stop responding for a while, with no output. Don't close the window; it should catch up and then display information about the process of installing the addon eventually. Just wait it out a bit.

With the previous command, we've enabled the `dns` addon. As I've mentioned, there are other addons available, but that's enough for now. At this point, just keep in mind that you can extend MicroK8s with addons, so it may be worth exploring in more detail later on if you wish. A list of addons is included in a link at the end of this chapter.

As mentioned earlier, if you're using Linux to run MicroK8s, then you would've used `sudo` with the `microk8s` command we used earlier. If you'd like to remove the requirement of using `sudo` on Linux, you can do so by adding your user to the `microk8s` group. That can be done with the following command:

```
sudo usermod -a -G microk8s jay
```

After you log out and then log in again, you should be able to run `microk8s` commands without `sudo`.

As we'll discover as we proceed through the rest of the chapter, the `kubectl` command is generally used to interact with a Kubernetes cluster and manage it. `kubectl`, short for **Kube Control**, is the standard utility you use to perform many tasks against your cluster. We'll explore this more later on. But specific to MicroK8s, it's important to understand that it uses its own version of `kubectl` that's specific to it. So with MicroK8s, you would run `microk8s kubectl` instead of simply `kubectl` to interact with the cluster. Having a separate implementation of `kubectl` with MicroK8s allows you to target an actual cluster (the `kubectl` command by itself) or specifically your installation of MicroK8s (prefix `kubectl` commands with `microk8s`), so one won't conflict with the other. As we work through the chapter, I'll call out `kubectl` by itself when we get to the actual examples, so it will be up to you to remember to use `microk8s` in front of such commands as needed.

Now that we have our very own installation of MicroK8s on our laptop or desktop, we can proceed through the examples in this book. You can skip ahead to the *Deploying containers via Kubernetes* section later in this chapter to begin working with Kubernetes. In the next section though, we'll take a look at setting up a Kubernetes cluster manually without MicroK8s, which is closer to the actual process you'd use in an actual production implementation.

## Setting up a Kubernetes cluster

In the previous section, we set up MicroK8s, which provides us with a Kubernetes cluster on a single machine, which is great for testing purposes. That might even be all you need in order to learn Kubernetes and see how it works. If you can, I still recommend setting up a cluster manually, which will give you even more insight into how the individual components work.

When it comes to a production installation in an actual data center, having Kubernetes installed on multiple servers is commonplace. Typically, one of them will act as the master node, and then you can add as many worker nodes as you need. As your needs expand, you can add additional servers to provide more worker nodes to your cluster. Setting up a master Kubernetes node and then individual workers is a great way to see the actual relationship in action. And that's exactly what we're going to do in this section.

As I walk you through the process, I'm going to do so utilizing three VMs. The first will be the master, and the remaining two will be workers. On your end, it doesn't really matter how many workers you decide to go with. You can have a single worker, or a dozen—however many you'd like to set up is fine. If you do set up more nodes than I do, then you would simply repeat the commands in this section for the additional nodes. For the nodes on your end, feel free to use whatever combination of physical or VMs makes sense and fits within the resources of the equipment you have available. Feel free to utilize Raspberry Pis if you have any available.

Before we get started, you'll want to have already installed Ubuntu Server on each machine and install all of the updates. It's also a good idea for you to create a user for yourself if you haven't already done so. The remainder of this section will assume you've already set up Ubuntu on each. After you've installed Ubuntu on everything, it's a good idea to also configure the hostnames on each node as well to make it easier to tell them apart. For example, on the node that I'm intending to use as the master, I'm going to set the hostname as `k8s-master`. For the workers, I'll name them `k8s-worker-1` and `k8s-worker-2`. You can name yours in whatever theme makes sense, but the reason I specifically mention it is to make sure that you name them something, as by default they'll show the same hostname in the command prompt, and that may be confusing.

Now that you have everything you need, let's set up Kubernetes!

## Configuring Docker

In this section, the commands that I'm going to have you run should be executed on each of the instances you plan on using with Kubernetes, regardless of whether or not you're working with the intended master or a worker; these commands need to be run on each. There will be commands specific to the master and the workers later on, but I'll mention the intended target if it's something I'd like you to do on specifically one or the other.

The first change you should make is specific to the Raspberry Pi. If your node is *not* a Raspberry Pi, then you should skip this. Even though Ubuntu is largely the same on each supported device, there are a few differences here and there, and this is one. Essentially, we need to make some changes to the boot process to enable various flags that are required for Kubernetes to function on a Pi. What you'll have to do is edit the `/boot/firmware/cmdline.txt` file and add some additional options:

```
cgroup_enable=cpuset cgroup_enable=memory cgroup_memory=1 swapaccount=1
```

Those options should be added to the end of the existing line within that file and not in a new line. When you open the file, you'll see existing options there, just press *Space* at the end and then input the additional values above. When you've finished editing the file, there should still be only one line.

> This particular change is the last that we'll add that's specific to the Raspberry Pi. The remaining changes we'll be making will be the same regardless of the type of device you're working with.

Next, we need to install Docker. An install script for users to be able to quickly install the required software is available at `get.docker.com`. The following command will download the installation script for Docker and run it:

```
curl -sSL get.docker.com | sh
```

Once the process is complete, you'll see output that gives you a tip regarding adding your local user to the `docker` group, which will allow you to manage containers without switching to the `root` account or using `sudo`. It's up to you whether or not you'd like to do that; at this point in our journey you should be able to make that decision yourself. If you would like to do so, you can use the `usermod` command to add your user to the group, as we've done several times in the past.

I am using an account named `jay` on my end, so for me the command I would need to run in order to add my user to the `docker` group becomes this:

```
sudo usermod -aG docker jay
```

The next order of business is for us to edit the `/etc/docker/daemon.json` file, which doesn't exist yet. This file allows us to tune the Docker daemon with additional parameters. The following content can be inserted into the file:

```
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
```

In this file, we're setting the `cgroupdriver` to `systemd`. The concept of **cgroups** is beyond the scope of this chapter, but as a short summary, it's a built-in feature of the Linux kernel that controls access to system resources (such as network, CPU, RAM, and so on) by processes that are running on our server. In this case, `systemd` is the init system that Ubuntu uses to control processes, so we want to make sure that Docker uses it to control access to cgroups. We're also setting a maximum size for the logs, in our case `100m`.

Moving on, we will need to edit another file, this time `/etc/sysctl.conf`. This time, the file will exist. Inside the file, you'll find the following line:

```
#net.ipv4.ip_forward=1
```

All you'll need to do at this point is uncomment that line (remove the `#` symbol in front of it) and then save the file. Doing this enables routing between network interfaces, which is disabled by default.

Since we've changed several things, we should reboot each of our nodes:

```
sudo reboot
```

Once your nodes finish starting back up, we can proceed to the next step.

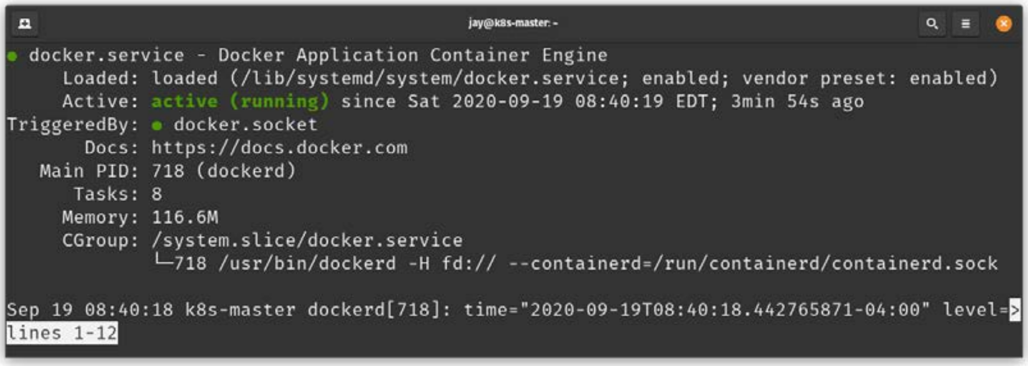
## Testing Docker

Before we continue and set up Kubernetes, we should make sure that Docker functions as it should. It's a good idea to test Docker on each node, to make sure we didn't make any mistakes while setting it up.

First, the Docker daemon should be running on each node:

```
systemctl status docker
```

We should see a status of `active (running)`:



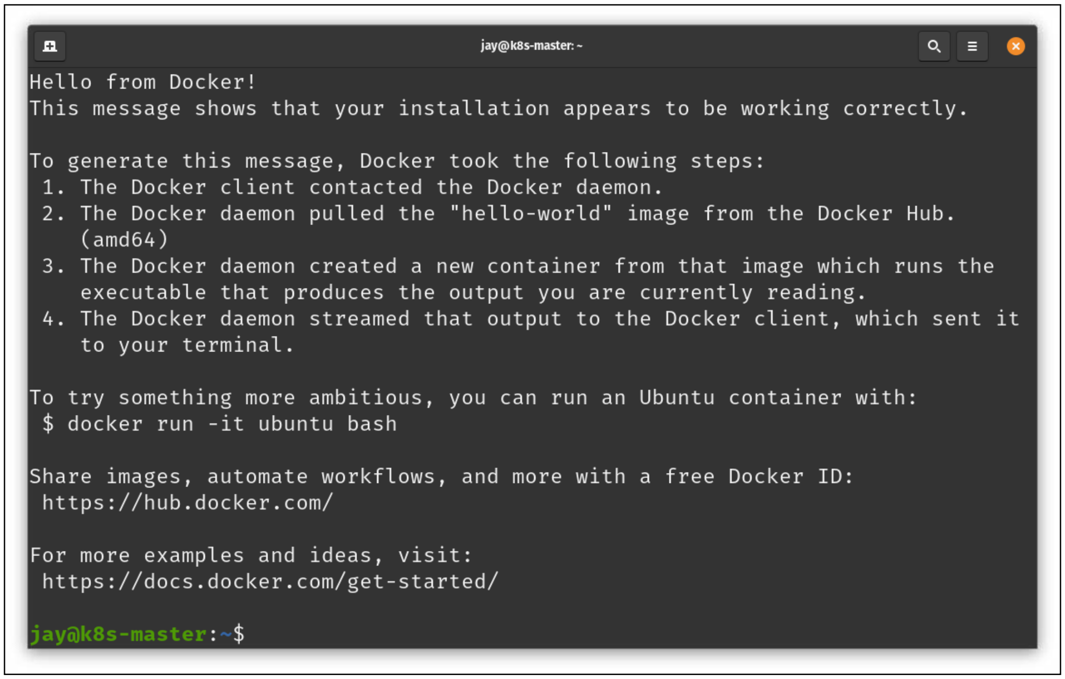
```
jay@k8s-master:~
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2020-09-19 08:40:19 EDT; 3min 54s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 718 (dockerd)
      Tasks: 8
     Memory: 116.6M
     CGroup: /system.slice/docker.service
             └─718 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

Sep 19 08:40:18 k8s-master dockerd[718]: time="2020-09-19T08:40:18.442765871-04:00" level=>
lines 1-12
```

Figure 18.10: Checking the status of the Docker daemon

To further test Docker, we can run a special container that's specifically intended to verify that it's working properly. This container is known as `hello-world` and will give us a helpful message indicating that Docker is working if it's able to run the container. We can run `hello-world` with the following command:

```
docker run hello-world
```

The container doesn't exist locally (at least not yet), so the command will first download the container image, and then run it. We should see output like this:



```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

jay@k8s-master:~$
```

Figure 18.11: Running a test container to verify that Docker is working properly

Did it work? If you've run into any problems, make sure that you've followed all of the steps up until now to get Docker installed, verified that it's running, and that you've also added your user to the `docker` group.

As long as you were able to run the test container, we're finished when it comes to setting up Docker, and we can now start installing Kubernetes.

## Installing Kubernetes

In this subsection, we're going to focus on setting up Kubernetes, which will take Docker to the next level and allow us to schedule containers to run on our worker nodes. Unless I mention otherwise, each of the commands in this section should be run against each node.

The first thing we should do is disable **swap** on each node. This may seem surprising, as swap can be helpful to us if our server runs out of memory. With regards to Kubernetes, it's preferred by design to have us utilize resource limits to prevent containers from saturating available memory, so swap can actually be counter-intuitive in our case. The following command will turn off swap:

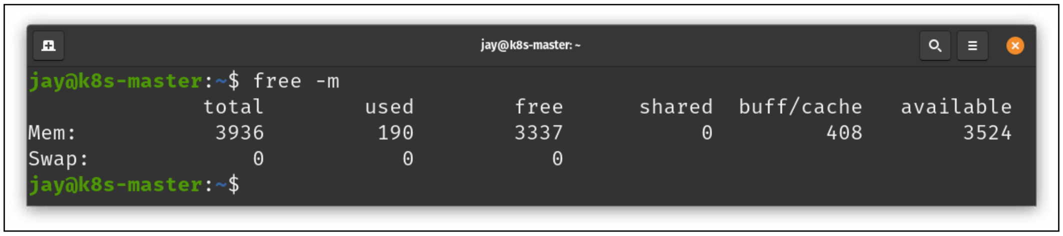
```
sudo swapoff -a
```

We should run that command on each node, to ensure that none of them have swap in use. However, the `swapoff -a` command is temporary, as soon as we reboot a server after entering that command, swap will be re-enabled. We should also make the change permanent on each node as well. To do that, we can edit the `/etc/fstab` file and comment out the line that enables swap. This may vary a bit from distribution to distribution, but for me, the appropriate line in that file looks like this:

```
/swap.img   none   swap   sw    0     0
```

On your end, you may have different syntax for the line in the file that enables swap. For example, if you've upgraded from a previous version of Ubuntu (before it switched to using swap files) you may have a dedicated partition as a swap partition. Either way, comment out the line that references swap. Make sure that you don't comment out or change any other line.

After all is said and done, you can enter the `free -m` command to verify that you don't have swap enabled. If disabled correctly, you should have all zeros for `Swap` in the output. For me, it looks like the following:



```
jay@k8s-master:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:           3936         190        3337           0         408        3524
Swap:             0           0           0
jay@k8s-master:~$
```

Figure 18.12: Checking to make sure swap is disabled

Next, we should add the appropriate repository so we can have access to the required packages for installing Kubernetes. To add the repository, we'll first add the key for the repository so our server will accept it as a trusted resource:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo
apt-key add -
```

To add the actual repository itself, we will create a file for it in the `/etc/apt/sources.list.d` directory. We can use an editor to open a new file at that path:

```
sudo vim /etc/apt/sources.list.d/kubernetes.list
```

Inside that file, we can enter the following line:

```
deb http://apt.kubernetes.io/ kubernetes-xenial main
```

What you may notice is that the line references `xenial` instead of the actual codename for Ubuntu 20.04, which is `focal` (short for Focal Fossa). At the time of writing, there is no dedicated Kubernetes repository for Ubuntu 20.04 yet, but the process will still work just fine. It's possible that by the time you're reading this, there will be a dedicated repository for 20.04. But for now, we can use the line mentioned above for our repository file.

With both the repository and the key installed, we can update our repository index to make the packages within the repository available to us:

```
sudo apt update
```

Next, we can install the packages required for Kubernetes:

```
sudo apt install kubeadm kubectl kubelet
```

We're installing three packages, `kubeadm`, `kubectl`, and `kubelet`:

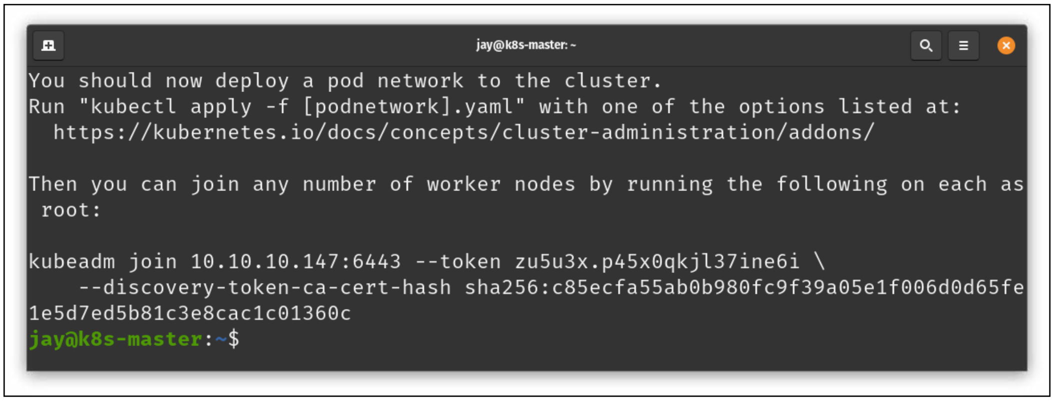
- **kubeadm**: The `kubeadm` package gives us tools we can use to "bootstrap" our cluster. We can use this tool to initialize a new cluster, join a node to an existing clusterand upgrade the cluster to a newer version.
- **kubectl**: This package provides us with the Kubernetes command-line tool, `kubectl`. We will use this tool to interact with our cluster and manage it.
- **kubelet**: The Kubernetes `kubelet` acts as an agent that facilitates communication between nodes. It also exposes API endpoints that can be used to enable additional communication and features.

Unlike all of the previous commands we've run through so far while setting up our cluster, the following will be entered and ran only on the node that you've designated as the master:

```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

The previous command will initialize the cluster, and also assign it a **Pod Network**. The Pod Network is an internal-only network that is not accessible from the outside. This means that you will not normally be able to ping an IP address within the Pod Network from another node on your LAN. We'll talk more about that later. You'll want to avoid the temptation of customizing the IP address range in the previous command and use it as-is. While it may be useful to give it a custom IP address scheme, and it's certainly possible to do so, but if you do change it you'll have communication issues within the cluster that are beyond the scope of this chapter to fix. For now, I recommend accepting the above defaults.

If the initialization process is successful, you'll see a `join` command printed to the terminal at the end. If you do see it, then congratulations! You've successfully set up a Kubernetes cluster. It's not a very usable cluster yet, because we haven't added any worker nodes to it at this point. But you do, by definition, have a Kubernetes cluster in existence at this point. On my master node, I see the following output:

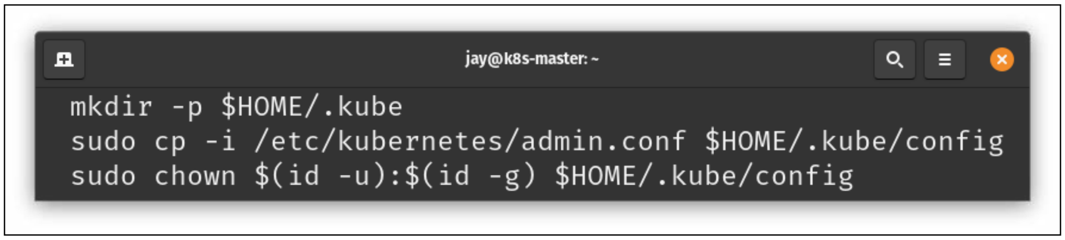


```
You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as
 root:

kubeadm join 10.10.10.147:6443 --token zu5u3x.p45x0qkjl37ine6i \
    --discovery-token-ca-cert-hash sha256:c85ecfa55ab0b980fc9f39a05e1f006d0d65fe
1e5d7ed5b81c3e8cac1c01360c
jay@k8s-master:~$
```

Figure 18.13: Successful initialization of a Kubernetes cluster, showing a join command

The `join` command that's shown to you after the process is complete is very special—you can copy that command and paste it into the command prompt of a worker node to join it to the cluster, but don't do that just yet. For now, copy that command and store it somewhere safe. You don't want to allow others to see it, because it contains a hash value that's specific to your cluster and allows someone to join nodes to it. Technically, showing my `join` command with the hash value is the last thing I should do in a book that will be seen and read by many people, but since this is just a test cluster with no actual value, I don't mind you seeing it.

Also, in the same output as the `join` command, are three additional commands we should run on the master node. If you scroll up in your terminal window to before the `join` command, you should see output similar to the following:



```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

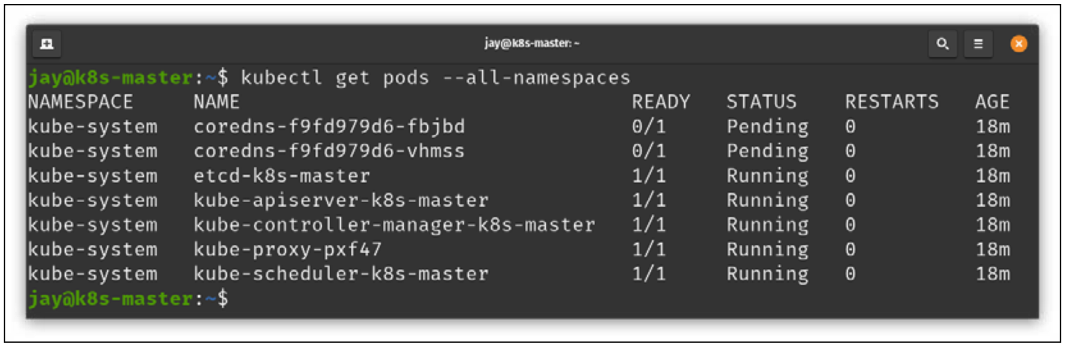
Figure 18.14: Output of the initialization process for Kubernetes showing additional commands to run

For those three commands, you can copy and paste them as-is right back into the terminal, one by one. Go ahead and do that. Essentially what you're doing is creating a local config directory for `kubectl` right in your user's home directory. Then, you're copying the `admin.conf` file from `/etc/kubernetes` and storing it in the newly created `.kube` folder in your home directory with a new name of `config`. Finally, you're changing ownership of the `.kube` directory to be owned by your user. This will allow your user account to manage Kubernetes with the `kubectl` command, without needing to be logged in as `root` or use `sudo`.

Kubernetes itself consists of multiple **Pods**, within which your containers will run. We haven't deployed any containers yet, we'll do that later. But even though we didn't deploy any containers ourselves, there are some that are actually running already. See for yourself:

```
kubectl get pods --all-namespaces
```

On my end, I see the following:



```
jay@k8s-master:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                   READY   STATUS    RESTARTS   AGE
kube-system   coredns-f9fd979d6-fbjbd                0/1     Pending   0          18m
kube-system   coredns-f9fd979d6-vhmss                0/1     Pending   0          18m
kube-system   etcd-k8s-master                        1/1     Running   0          18m
kube-system   kube-apiserver-k8s-master              1/1     Running   0          18m
kube-system   kube-controller-manager-k8s-master     1/1     Running   0          18m
kube-system   kube-proxy-pxf47                       1/1     Running   0          18m
kube-system   kube-scheduler-k8s-master              1/1     Running   0          18m
jay@k8s-master:~$
```

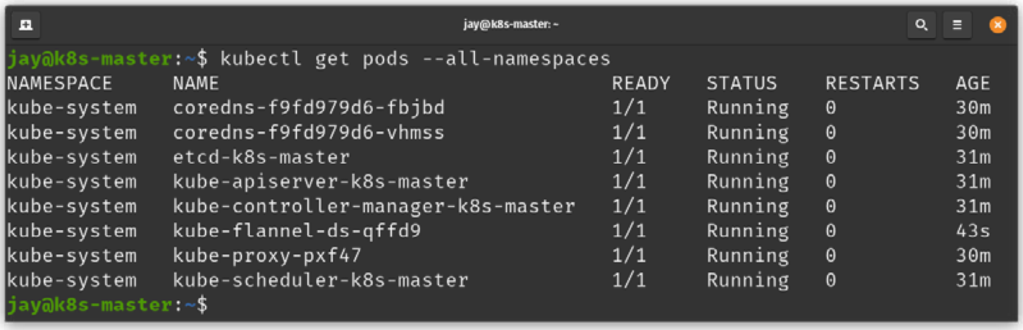
Figure 18.15: Checking the status of pods running in our Kubernetes cluster

Notice that the namespace for each of these Pods is `kube-system`. This is a special namespace, where Pods related to the Kubernetes cluster itself will run. We didn't explicitly ask for these to run, they're run as part of the cluster and provide essential functionality. Another important column is `STATUS`. In the screenshot, we see that most of them are `Running`. The first two, though are in a state of `Pending`, which means that they're not quite ready yet. This is actually expected, we haven't deployed the Pod Network yet, which is required for the `coredns` Pods to function. Other Pods may also show a status of `Pending`, and those should automatically switch to `Running` when they finish setting themselves up. How long this takes depends on the speed of your hardware, but it should only take a few minutes.

Let's deploy the Pod Network so we can make the `coredns` Pods happy:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/
master/Documentation/kube-flannel.yml
```

When we check the status again, we should see an additional Pod running (`kube-flannel-ds`), and the Pods that had a state of `Pending` previously should now be `Running`:



```
jay@k8s-master:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                 READY   STATUS    RESTARTS   AGE
kube-system   coredns-f9fd979d6-fbjbd              1/1     Running   0          30m
kube-system   coredns-f9fd979d6-vhmss              1/1     Running   0          30m
kube-system   etcd-k8s-master                      1/1     Running   0          31m
kube-system   kube-apiserver-k8s-master            1/1     Running   0          31m
kube-system   kube-controller-manager-k8s-master   1/1     Running   0          31m
kube-system   kube-flannel-ds-qffd9                1/1     Running   0          43s
kube-system   kube-proxy-pxf47                     1/1     Running   0          30m
kube-system   kube-scheduler-k8s-master            1/1     Running   0          31m
jay@k8s-master:~$
```

Figure 18.16: Checking the status of Pods running in our Kubernetes cluster again

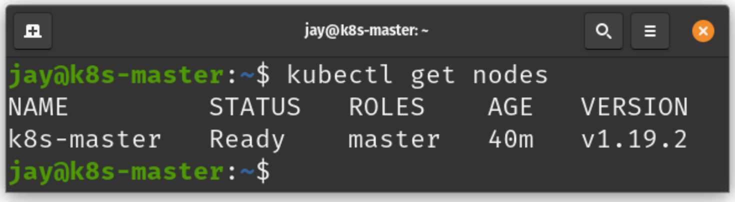
So, what exactly did we do? **Flannel** is a networking layer that can run on Kubernetes. Networking is required for Kubernetes to function, as Pods within the cluster need to be able to communicate with one another. There are multiple different types of networking models you can implement in Kubernetes, Flannel is just one available option. If you're using a cloud provider, such as AWS, then the networking model is typically chosen for you. Since we're building a cluster from scratch, we have to choose a networking model—Kubernetes itself doesn't come with one.

Setting up networking in a cluster that was manually created can be quite an involved task, Flannel is easy to set up (we simply deploy it) and its defaults meet the needs of Kubernetes and will get us up and running quickly. A full walk-through of networking models in Kubernetes is beyond the scope of this chapter, but Flannel is good enough for us.

Next, it's time to watch the magic happen and join worker nodes to our cluster. We can check the status of all of our nodes with the following command:

```
kubectl get nodes
```

Right now though, we only have the master showing up in the output since we haven't yet added any nodes:



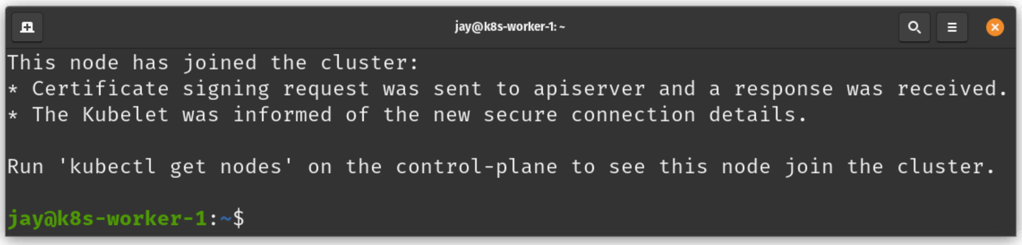
```
jay@k8s-master:~$ kubectl get nodes
NAME         STATUS   ROLES    AGE   VERSION
k8s-master   Ready    master   40m   v1.19.2
jay@k8s-master:~$
```

Figure 18.17: Checking the status of the nodes within the cluster

To add a worker to our cluster, we can enter the `join` command we saw earlier on a node designated as a worker. If you recall, the `join` command was shown in the terminal when we first initialized our cluster. The command will look something like the following, which I've shortened a bit to fit on this page (I've added `sudo` to the beginning):

```
sudo kubeadm join 10.10.10.147:6443 --token zu5u3x.p45x0qkjl37ine6i \
    --discovery-token-ca-cert-hash sha256...1360c
```

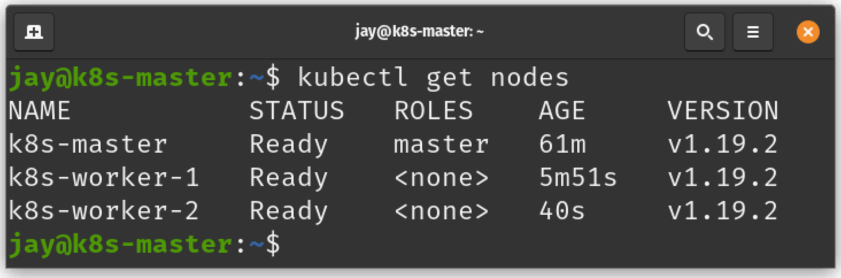
For you, the command will be very different. The IP address in the command is for the master, which will no doubt be different on your end. The hash value will be different as well. Basically, just copy the `join` command you were provided while initializing the cluster on the master and paste it into each of your worker nodes. You should see a message in the output that the node was successfully added to the cluster:



```
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

jay@k8s-worker-1:~$
```

Figure 18.18: Successfully adding a worker node to the Kubernetes cluster

After you've run the join command on each of your worker nodes (on however many you decided to create), you can run the `kubectl get nodes` command again on the master and verify the new nodes appear on the list. I added two nodes, so I see the following output on my end:



```
jay@k8s-master:~$ kubectl get nodes
NAME           STATUS   ROLES    AGE     VERSION
k8s-master     Ready    master   61m     v1.19.2
k8s-worker-1   Ready    <none>   5m51s   v1.19.2
k8s-worker-2   Ready    <none>   40s     v1.19.2
jay@k8s-master:~$
```

Figure 18.19: Output of kubectl showing worker nodes now added to the cluster

At this point, our cluster exists and has one or more worker nodes ready to do our bidding. The next step is to actually use the cluster and deploy a container. That's exactly what we'll do in the next section.

## Deploying containers via Kubernetes

Now it's time to see our work pay off, and we can successfully use the cluster we've created. At this point, you should have either set up MicroK8s, or manually created a cluster as we've done in the previous section. In either case, the result is the same: we have a cluster available that we can use to deploy containers.

Keep in mind that if you're using MicroK8s, you'll need to prepend `microk8s` in front of `kubectl` commands. I'll leave it up to you to add `microk8s` to the front of such commands as you go along, if you're using MicroK8s.

Kubernetes utilizes files created in the YAML format to receive instructions. Does that sound familiar? In *Chapter 15*, *Automating Server Configuration with Ansible*, we worked with YAML files as that's the format that Ansible playbooks are written in. YAML isn't specific to Ansible; it's used with many different applications and services, and Kubernetes will also recognize the YAML format to contain instructions for how to deploy something. Here's an example YAML file to get us started:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-example
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: linuxserver/nginx
      ports:
        - containerPort: 80
          name: "nginx-http"
```

Before I show you how to run it, let's walk through the file and understand what's going on. First, we're identifying the API version we intend to use, and then we're setting the `kind` of deployment we intend to set up. In this case, we're deploying a `pod`, which is what our containers run inside of in Kubernetes. One or more containers can run in a Pod, and a worker node can run one or more Pods at the same time.

Next, we add some `metadata`. Metadata allows us to set special parameters specific to our deployment. The first item of metadata we're customizing is the `name`, we're naming the Pod `nginx-example` in this case. We're also able to set up labels with metadata, which is a "name: value" key pair that allows us to add additional values to our Pod that we can refer to later. In this case, we're creating a label called `app` and setting it to `nginx`. This name is arbitrary; we could call it `potato` if we wanted to. Setting this to `nginx` is a descriptive label that makes it obvious to someone else what we are intending to run here.

Moving on, the `spec` section allows us to specify what exactly we want to run in our Pod, and how we want it to run. We want to run a container in our Pod, and specifically a container that we'll name `nginx`, which we'll retrieve from a registry called `linuxserver`, and we're requesting a container from that registry by the name of `nginx`.

The registry we're fetching the container image from deserves some extra explanation. This registry in particular is located at `https://linuxserver.io`, which is a special service that makes container images available for us to download and use. The site has a documentation section that gives us information about each of the container images they offer. Why use the `linuxserver.io` registry? The reason is that their service makes available various container images that support a variety of architectures, including x86 as well as ARM. The latter is especially important, because if you're using Raspberry Pi units for your cluster, they won't be able to utilize container images created for x86. If you do attempt to run a container image that does not support ARM, then the container will fail to launch on a Pi. Since `linuxserver.io` makes container images available for multiple architectures, they should work fine regardless of the type of device you decided to use for your cluster. Whether you're using an x86 physical server or VM for your worker node, the `nginx` container we're retrieving from that registry should function just fine.

In the last few lines, we're setting up a container port of `80`, which is the standard for a web server. This is the default port that NGINX listens on when it runs, and NGINX is what we're intending to run inside our container.

Before we continue, there's a bit of difference regarding how you deploy resources to a cluster within MicroK8s compared to a cluster that was set up manually on actual servers. The commands going forward will assume you've set up a cluster manually. If you're using MicroK8s on Windows or macOS, you'll need to copy any deployment files you create into the MicroK8s VM that's created as part of the installation of MicroK8s. If you try to save a file locally and deploy it as we're about to do in the next paragraph, it will fail, because the file will not be found on the VM. To copy a deployment file to the MicroK8s VM to enable you to deploy it, you can use the following command to copy the file into the VM first:

```
multipass transfer <filename.yml> microk8s-vm:
```

As we go along, keep in mind that whenever we're deploying a file, you'll have to make sure to transfer it first. Also, a manually created cluster uses the `kubectl` command, while MicroK8s requires you to prefix `kubectl` with `microk8s`, so such commands become `microk8s kubectl` instead of just `kubectl`.

Let's go ahead and deploy the container to our cluster. Assuming you named the YAML file as `pod.yml`, you can deploy it with the following command on the master node:

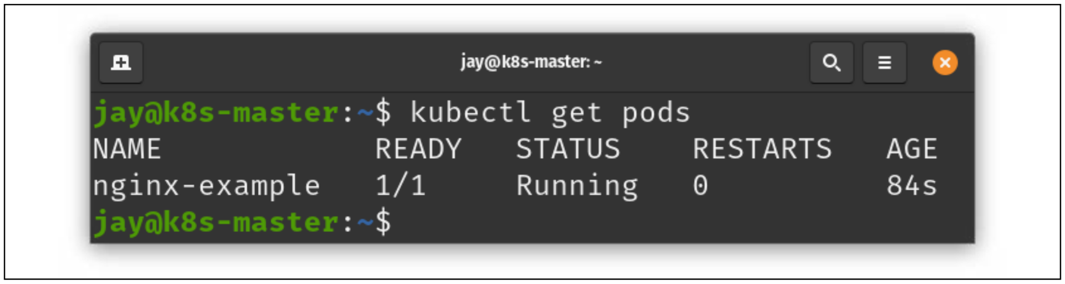
```
kubectl apply -f pod.yml
```

As mentioned previously, the `kubectl` command allows us to control our cluster. The `-f` option accepts a file as input, and we're pointing it to the YAML file that we've created.

Once you've run the command, you'll be able to check the status of the deployment with the following command:

```
kubectl get pods
```

For me, I see the following when I run it:



```
jay@k8s-master:~$ kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
nginx-example   1/1     Running   0          84s
jay@k8s-master:~$
```

Figure 18.20: Checking the status of a container deployment

From the output in my case, I can see that the process was successful. The `STATUS` is `Running`. What we don't see is what worker node the Pod is running on. It's nice to know that it's `Running`, but where? We can get more information by adding the `-o wide` option to the end of the command:

```
  kubectl get pods -o wide
```

The output contains much more information, so much that a screenshot of it won't fit on this page. The focus though, is that among the extra fields we have with this version of the command is the `node` field, which shows us which worker node is handling this deployment. We'll even see an IP address assigned to the Pod as well.

We can use the IP address shown for the Pod to access the application running inside the container. In my case, my Pod was given an IP address of `10.244.1.2`, so I can use the `curl` command to access the default NGINX web page:

```
curl 10.244.1.2
```

The output that's shown in the terminal after running that command should be the HTML for the default NGINX web page. Note that the `curl` command may not be available in your Ubuntu installation, so you may need to install the required package first:

```
sudo apt install curl
```

We have one potential issue though: if you want to be able to access an application running inside the cluster on a machine other than the master or worker nodes, it won't work. By default, there's nothing in place to route traffic from your LAN to your cluster. This means that in my case, the IP address of `10.244.1.2` that was given to my Pod was provided to it by the Pod Network, the router on my network doesn't understand that IP address scheme so trying to access it from another machine on the LAN will fail. What's interesting is that you can access the application from any other node within the cluster. In my case, the NGINX Pod is running on the first worker node. However, I can actually run the previous `curl` command from worker #2 even though the Pod isn't running there, and I'll get the same exact output. This works because the Pod Network is cluster-wide; it's not specific to any one node. The IP address of `10.244.1.2` is unique to the entire cluster, so if I run another container it won't receive that same IP address, and every node within the cluster knows how to internally route to IPs within that network.

Not allowing outside devices to access applications within the cluster is great for security purposes. After all, a hacker can't break into a container that they can't even route to. But the entire point of running a Kubernetes cluster is to make applications available to our network, so how do we do that? This can be a very confusing topic for newcomers, especially when we're setting up a cluster manually. If we're using a cloud service such as AWS or Google Cloud, they add an additional layer on top of their Kubernetes implementation that facilitates routing traffic in and out of the cluster. Since we set up our cluster manually, we don't have anything like that in place.

When it comes to building services and networking components designed to handle networking to bridge our LAN and Kubernetes cluster, that's an expansive topic that can span a few chapters in and of itself. But a simple solution for us is to create a **NodePort Service**. These are two new concepts here, a Service as well as NodePort. When it comes to a Service, a Pod isn't the only thing we can deploy within our cluster. There are several different things we can deploy, and a Service is a method of exposing a Kubernetes Pod, and a NodePort is a specific type of service that gives us a specific method for facilitating a way of accessing it. What NodePort itself does is expose a port running inside a Pod to a port on each cluster node.

Here's a file we can deploy to create a NodePort Service for our Pod:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-example
spec:
  type: NodePort
  ports:
    - name: http
      port: 80
      nodePort: 30080
      targetPort: nginx-http
  selector:
    app: nginx
```

As you can see, the `kind` is `Service`; as this time, we're deploying a service to our cluster to complement the Pod we've deployed previously. The `type` of service is `NodePort`, and we are mapping port `80` within the Pod to port `30080` in our cluster. I chose port `30080` arbitrarily. With NodePort, we can utilize ports `30000` through `32767`. The `selector` in this file is set to `nginx`, which is the same selector we used while creating the Pod. Selectors allow us to "select" Kubernetes resources via a name we assign them, to make it easier to refer to them later.

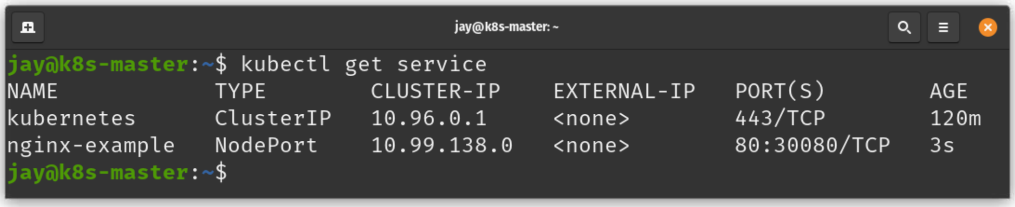
Let's deploy our service:

```
kubectl apply -f service-nodeport.yml
```

Assuming we've typed everything in the YAML file for the service properly, we can retrieve the status of services running within our cluster with the following command:

```
kubectl get service
```

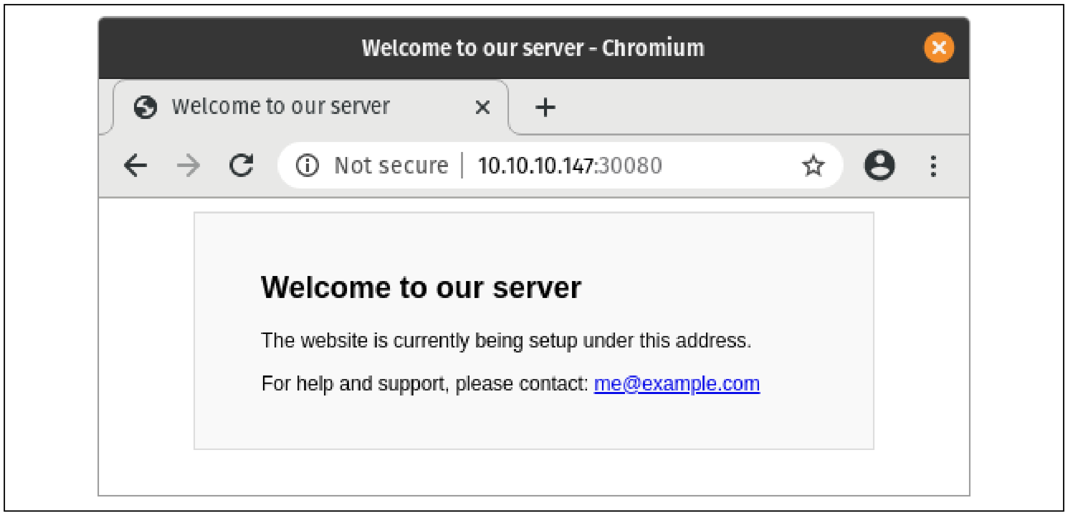
If everything has gone well, you should see output similar to the following:



```
jay@k8s-master:~$ kubectl get service
NAME            TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1     <none>        443/TCP        120m
nginx-example   NodePort    10.99.138.0   <none>        80:30080/TCP   3s
jay@k8s-master:~$
```

Figure 18.21: Checking the status of a service deployment

In the output, we should see the status of our `service` deployment, which is the second line in the screenshot. We can also see the port mapping for it, showing that port `30080` should be exposed to the outside. To test that it's actually working, we can open a web browser on a machine that is within our LAN, and it should be able to access the master's IP address at port `30080`:



Welcome to our server

The website is currently being setup under this address.

For help and support, please contact: me@example.com

Figure 18.22: Accessing a web page served by an NGINX container in a cluster from within a web browser

Another interesting benefit is that we didn't actually have to use the IP address of our master node, we can use the IP address of a worker node as well and see the same default page. The reason this works is that the mapping of port `30080` to port `80` inside the Pod is cluster-wide, just as the internal Pod Network is also cluster-wide. Accessing a resource on one is the same as accessing the same resource on any other, as the request will be directed toward whichever node is running the Pod that matches the request.

When it comes to removing a Pod or a Service, assuming you want to decommission something running in your cluster, the syntax for that is fairly straightforward. To remove our `nginx-example` Pod, for example, you can run this:

```
kubectl delete pod nginx-example
```

Similarly, to delete our service, we can run this:

```
kubectl delete service nginx-example
```

At this point, you not only have a working cluster, but you also have the ability to deploy containers to it and set them up to be accessible from outside the Pod Network. From here, I recommend you practice with this a bit and attempt to run additional containers and apps to have a bit of fun with it.

## Summary

In this chapter, we took containerization to the next level and implemented Kubernetes. Kubernetes provides us with orchestration for our containers, enabling us to more intelligently manage our running containers and implement services. Kubernetes itself is a very expansive topic, and there are many additional features and benefits we can explore. But for the goal of getting set up on Kubernetes and running containers with it, we did what we needed to do. I recommend that you continue studying Kubernetes and expand your knowledge, as it's a very worthwhile subject to dig deeper into.

Speaking of subjects that are worthwhile to learn, in the next chapter, we're going to learn how to deploy Ubuntu in the cloud! Specifically, we'll get started with Amazon Web Services, which is a very popular cloud platform.

## Further reading

- MicroK8s website: `https://microk8s.io/`
- MicroK8s addons: `https://microk8s.io/docs/addons`

# 19
# Deploying Ubuntu in the Cloud

Up until now, in each chapter, we've been working with an instance of Ubuntu installed on either a local virtual machine, a physical computer or server, or even a Raspberry Pi. We've learned how to deploy Ubuntu on such devices, and we've even gone as far as deploying virtual machines as well as containers. These on-premises devices have served us well, but the concept of **cloud computing** has already become quite popular, even more so since the previous edition of this book. In this chapter, we're going to take a look at running Ubuntu in the cloud. Specifically, we'll deploy an Ubuntu instance on **Amazon Web Services** (**AWS**), which is a very popular platform for cloud computing. While we won't go into extreme detail on AWS (it's an extremely large and complex platform), you'll definitely get a feel for what it's like to deploy resources in the cloud, which will be more than enough to get you started.

This exploration will involve the following topics:

- Understanding the difference between on-premises and cloud infrastructure
- Important considerations when considering cloud computing as a potential solution
- Becoming familiar with some basic AWS concepts
- Creating an AWS account
- Choosing a region
- Deploying Ubuntu as an AWS EC2 instance

- Creating and deploying Ubuntu AMI images
- Automatically scaling Ubuntu EC2 deployments with Auto Scaling
- Keeping costs down: understanding how to save money and make cost-effective decisions
- Taking the cloud further: additional resources to grow your knowledge

Since cloud computing is something of a different mindset than we're used to, we'll first go through a few sections that are dedicated to helping us understand the difference, as well as some of the considerations we should make before choosing to implement Ubuntu in the cloud. In the next section, we'll explore how the concept of cloud computing differs from on-premises hardware.

## Understanding the difference between on-premises and cloud infrastructure

As mentioned at the very beginning of this chapter, we've been solely utilizing on-premises Ubuntu installations thus far. Even if we're running Ubuntu on a virtual machine in our data center, it's still considered an on-premises installation even when it's not on physical hardware. In short, an on-premises installation is something that resides locally with us, regardless of the type of server that serves as the foundation.

The first difference when it comes to cloud computing is somewhat obvious: it's the exact opposite of a resource being on-premises. With a cloud instance of Ubuntu, it's someone else's hardware that it runs on. Most of the time, we won't know what kind of server a cloud instance is running on—when we subscribe to the services of a cloud provider and pay a fee to run a server on that platform, we're able to access the operating system just like we would a virtual machine, with little knowledge of the underlying data center. Although utilizing a cloud instance will have some sort of recurring cost, we don't need to worry about monitoring hardware components nor replacing failed physical devices. With the cloud, that becomes someone else's problem.

I could actually end this section right there, as knowing the difference between on-premises and cloud is literally that simple (at a high level). But it also constitutes a change in mindset too, as well as how you manage cloud instances.

There are several cloud providers you can choose from, with **AWS**, **Google Cloud Platform** (**GCP**), and **Microsoft Azure**. Those are three very popular platforms. There are also **Virtual Private Server** (**VPS**) providers as well, such as **Digital Ocean** and **Linode**, among others.

Amazon also has a VPS service available as well, which they refer to as **Amazon Lightsail**. The concept of cloud computing has become more complex, since you have a variety of different services you can choose from.

First, let's understand the difference between a cloud service provider and a provider of VPSes. This distinction is rather complicated nowadays because the line between them seems to blur more and more as time goes on. Essentially, a cloud service provider such as AWS, GCP, or Azure provides you with a full menu of services, not just servers. This includes (but is definitely not limited to) services such as DNS, database instances, configuration management, software-defined networking, and more. AWS, for example, offers a service called **Route 53** that acts as a managed DNS service, and it allows you to not only configure DNS for your cloud network but even register domain names as well. AWS even offers a service for hosting servers to act as the backend infrastructure for online multiplayer games, which they call **GameLift**. In addition, cloud providers will also provide a managed service for running Kubernetes, so you won't have to manage the underlying cluster yourself. Other cloud service providers will also give you a full menu of services, often with different unique marketing terms.

VPS providers are very similar to cloud service providers in that they also offer "cloud" servers for a fee, but often with fewer services available to you and simpler flat pricing. For example, while VPS providers will all give you the ability to create an Ubuntu server on their platform, they won't typically have a service you can utilize to register domain names. Basically, their "menu" of available services is stripped down quite a bit from what full cloud providers will offer. However, as I've mentioned, the line between cloud and VPS providers blurs more and more, as VPS providers are starting to offer services that until recently were only available from cloud providers. For example, both Digital Ocean and Linode offer a managed Kubernetes service on which you can run containers, which previously was offered only on cloud platforms. VPS providers now also offer object storage as well, another service they didn't use to offer. Amazon now offers a VPS service of their own with Lightsail, for those of you that want to create cloud servers from a simpler service.

So, which should you choose? Should you sign up with a VPS provider such as Digital Ocean, Linode, or Lightsail? Or should you choose a full cloud provider such as AWS, GCP, or Azure? The decision comes down to why you want to create cloud infrastructure in the first place, and the features you desire to take advantage of. If you compare each offering, you'll see the features they offer to you, and you can choose according to which fits your use case best. Make sure to not only consider the features you need right now but also features you may need in the future.

Perhaps the most important question to ask right now is whether or not creating cloud resources is appropriate for your goals. That's exactly what we'll explore in the next section. We'll also explore some of the differences in mindset when it comes to cloud computing as well.

## Important considerations when considering cloud computing as a potential solution

Before choosing to sign up with a provider, it's important to first make sure that creating cloud resources is a good idea for you or your organization in the first place. Often, IT professionals can get so excited when it comes to a new trend that they may make the mistake of trying to use such a service even when it doesn't make sense to do so. Above all, as an administrator, it's important to utilize the best tool available for whatever it is that you wish to accomplish, instead of using a technology just because you're excited about it. Cloud computing is awesome for sure, but for some use cases, it's just not a good fit. This is similar to containers as well: containerization is an exciting technology but some applications just don't run well on that platform. It takes trial and error.

There are some considerable benefits when it comes to cloud computing. When it comes to physical servers, the hardware will fail eventually. It's not a matter of "if," it's a matter of "when." All hardware *will* fail eventually. And even if you have hardware that doesn't end up failing in the short term and lasts for a long time, it will be made obsolete by more powerful hardware that is more power-efficient. When it comes to managing physical servers, you'll need to replace the hardware eventually.

This is also true when it comes to cloud computing; the hardware such services run on will need to be replaced whenever it fails or becomes obsolete (whichever happens first). The difference is that when that happens, the liability on you (the administrator) is greatly reduced. You won't have to order a new server, replacement parts, or even pay attention to the hardware at all. It's solely the responsibility of the cloud service or VPS provider to keep track of that.

The trade-off, though, is potentially cost. Why I mention cost as a "potential" trade-off is because whether or not it's cheaper for your organization to purchase physical servers or pay a monthly fee for servers in the cloud comes down to which offers a better **Return on Investment** (**ROI**). To better understand this, consider an organization that utilizes only on-premises hardware for their data center with no cloud resources being used at all. Let's also assume that every 3 years or so, they have to replace some of the more critical servers with newer hardware; and every 5-10 years the lesser-important servers are replaced. There's also a need to pay for full-time administrators with specialties in managing physical hardware as well as maintaining a cooling system and making sure a room is available to designate as a data center. Add all of this up—how much is it costing the organization on average?

With physical infrastructure, you're paying an up-front cost to purchase servers every time you need to do so. With cloud computing, you never have to purchase physical server hardware, but instead you're paying a fee each and every month for the privilege of utilizing cloud infrastructure instead. When you add up the monthly costs for cloud resources, how much would that cost the organization? Would it cost more than running physical hardware, or less?

One very important consideration here is storage. If your organization doesn't store much data, then storage costs won't really be a concern. Many companies are perfectly satisfied with employees utilizing something like Google Drive for shared storage and won't need cloud storage. Some organizations, especially those that develop software, have *massive* storage needs. For these companies, there could be tens or even hundreds of terabytes of data being stored, as well as the bandwidth costs for the data going in and out of the company. If you were to attempt to migrate such a large amount of data to a cloud server provider, you would no longer have to manage storage hardware, but your costs for cloud storage would go up dramatically and might easily be much more expensive than it costs you to continue to use physical hardware.

Another consideration is stability. It's often argued that utilizing a cloud provider will result in a more stable infrastructure. That makes sense, considering that by using a cloud provider you're no longer responsible for managing hardware. There's definitely some truth to the claim that cloud resources are more stable, but it's not quite that simple.

Cloud providers will often advertise a high level of uptime, often measured in "9s." For example, AWS offers a general uptime of 99.99%, as of the date this book is being prepared for publishing. That sounds great, doesn't it? Before you get too excited about a cloud provider's uptime claim, it's important to also know what exactly they count as part of that uptime. If an upstream provider, such as a backend internet resource, goes down and service is completely unavailable, they may not count that and continue to claim the same amount of uptime. Massive outages of cloud providers are not uncommon, and a major outage of AWS occurred in 2017 where Amazon's S3 service suffered a major outage due to one of their engineers mistyping a command. In that situation, many services throughout the internet were unavailable to customers. So while it is true that a cloud service provider is less likely to suffer an outage than you are to witness hardware failures, it's important to keep in mind that outages are still possible. I don't mention this to scare you or steer you away from cloud providers, but merely to underscore the importance of backups and automation to get back up and running quickly during a failure. We can't ever assume any service (our own, or otherwise) is bulletproof.

Automation is a very important consideration with the cloud. If your resources encounter an issue, it's a really good idea to have an automated means of re-deploying your important services. On a physical server, you can set it up exactly as you want it and take an image of the hard drive in case the server fails. Cloud providers offer you essentially the same service, giving you the ability to create images of your important servers so you can redeploy them in the future should you need to do so. If you do go with a cloud provider, make sure you keep regular backups as well as images. And be sure to keep a copy of at least the most recent backups locally outside of the cloud provider, because if the cloud provider itself goes down then you also lose the backup. I can probably better summarize this by advising you to not become over-confident in the stability of your cloud provider. Always assume your infrastructure will fail eventually, regardless of where it's located.

The warnings I gave around the stability of cloud providers are not intended to scare you away from using them, but instead to steer you toward maintaining good hygiene with the cloud, just as you would with physical infrastructure. Cloud providers actually provide you with all the tools you need in order to build a stable infrastructure that's easily reproducible and recoverable. If you utilize those tools effectively, you shouldn't have anything to worry about. We'll explore these concepts further as we go along, but you can go as far as to have cloud servers recover *themselves* when a problem occurs. It all goes into how you architect your solution.

In the next section, we're going to explore some concepts specific to AWS, so we'll have a stronger foundation of knowledge to use for building an actual cloud solution later on in this chapter.

## Becoming familiar with some basic AWS concepts

As discussed earlier, AWS is one of several competing cloud service providers. For the purpose of this chapter, AWS was chosen because more than any other provider, the platform requires an administrator to adopt a completely different mindset when it comes to managing infrastructure. This different mindset is a healthy one even outside of AWS, so it represents a logical evolution at this point in our journey.

Up until now, we discussed server installations as essentially pets, meaning we want to keep them around, make sure they're healthy, and if something goes wrong, try to fix it. We want to keep our servers operational for as long as possible. We want to be able to rely on them, and that helps our organization—customers and clients appreciate using a website or service that is stable, with minimal or no downtime.

That last part, minimal downtime, doesn't change regardless of the mindset we use when managing our infrastructure. Downtime and service disruption is bad in this industry, and that's always going to be the case. The difference is what we do about it, and how we go about recovering from it. In fact, we can even try our best to automatically recover when we have a problem. If the customer never notices there was ever an issue, even better.

With AWS, it's best to consider our servers as disposable, not as pets. This may seem a bit surprising at first, but if we effectively use the tools we're provided, we can build infrastructure that is scalable. This concept in particular is known as **Auto Scaling** and is a very important aspect of AWS. With Auto Scaling, resources are automatically created and destroyed as demand increases and decreases. For example, let's say that your web server receives more visitors than normal, and its CPU is starting to max out. Auto Scaling would then bring up a new web server automatically, and with a load balancer, route clients between instances as needed. You can set the maximum number of instances that can be brought online, and then when the load decreases, the servers that were brought online to handle the load will automatically get deleted. This means that you will automatically have the number of servers in existence as you need at any given time.

Another level above Auto Scaling is **Auto healing**. Just as the name implies, auto healing means that if your server runs into a problem, it will automatically be disposed of and recreated. This is the ultimate goal of AWS, to have infrastructure that is not only scalable but is able to recover by itself. For example, perhaps you have a pair of web servers that handle requests from clients. One of them encounters some sort of issue and fails some sort of test that would imply that the server is unreliable. Such a test is known as a **health check**. With auto healing, a server that fails health checks is considered unhealthy and is deleted. With Auto Scaling, you also set a minimum number of servers for your application. If the number of servers falls below the minimum, then a new one is created to replace it. Depending on how you architect your solution, the customer may notice some degradation if the application is running on fewer servers, but that's only temporary. Everything will return back to normal when the new server comes online.

With AWS, Auto Scaling and auto healing are extremely important. I've seen many administrators make the mistake of not utilizing those features of the platform, and that should never be the case. Without such high availability features, you risk having your server go away at any moment. This may be a bigger problem than you think. AWS consists of physical servers all over the world, and just like any other physical server, hardware failures are possible. If your application is running on a physical host within an AWS data center that encounters a hardware failure, your server may get deleted when they go and replace it. With Auto Scaling, this isn't a problem. A new virtual server will be brought online to replace it.

But without Auto Scaling, this can result in manually having to rebuild your server. Don't worry though, we'll cover Auto Scaling in this very chapter, so you'll definitely understand how to implement it before we're done.

The concepts around high availability are not limited to AWS. Similar features exist on other cloud platforms as well. The main difference between competing cloud providers is the marketing terms they use for these features, but you can set up similar infrastructure on each. With AWS, they really focus on this aspect though, so it's important to learn. But even if you're not utilizing AWS in production at your organization, the concepts around being able to easily recover from disasters is still important and can still be implemented. If nothing else, you should at least consider implementing automation for rebuilding servers so you don't have to do so much manual work if a problem occurs.

Back to AWS: there are many services within the platform that you can use that will provide a variety of features. In fact, there are well over a hundred different services within the platform. Therefore, it's impossible for us to cover each in this chapter. It would be difficult to cover every service even in a book dedicated only to AWS. But don't let the number of services overwhelm you though, you'll only need to learn the services that are related to what you're trying to achieve with the platform. For example, if your organization doesn't develop multiplayer computer games, then learning the GameLift service will be of no benefit to you at all. In this chapter, we'll focus on the services required to get you up and running with a basic web server running in the cloud. The following services and terms will be discussed:

- **Virtual Private Cloud** (**VPC**): A VPC is a higher-level abstraction of your overall cloud network and resources. Each of the servers and related services you provision will fit within a VPC. You can think of a VPC as your overall network. In your organization, you may have various routers, gateways, firewalls, virtual machines, printers, or other network-connected devices. Your organization's network is essentially a VPC in AWS, a software version of a complete network.
- **Elastic Compute Cloud** (**EC2**): EC2 is the service within AWS that your virtual machines will run in. Individual virtual machines within AWS are referred to as EC2 instances. You can think of this service as the AWS-equivalent of VMware ESXi, Microsoft Hyper-V, Proxmox, or whatever virtual machine platform you're more familiar with. EC2 instances, just like a virtual machine, have memory and CPU allocated to them and run an operating system. We'll run Ubuntu in an EC2 instance before the end of the chapter.

- **Elastic Block Store** (**EBS**): As you learn more about AWS, you'll notice that even the simplest component seems to have a marketing buzzword attached. EBS provides block storage, essentially the same type of storage we've been working with all along. So in a nutshell, an EBS volume is a hard disk. When you create an EC2 instance, the operating system for your server will run from an EBS volume, and you can set the size of the volume accordingly. We'll work through this later in the chapter.
- **Elastic Load Balancer** (**ELB**): As you may be able to guess from the name, an ELB is the AWS equivalent of a load balancer and offers similar features. This allows you to have multiple EC2 instances serving your application, and you can create an ELB to route traffic between them. ELB is actually a feature of EC2 and not its own service.
- **Identity and Access Management** (**IAM**): IAM is the tool within AWS that you'll use to create and manage user accounts, determine user permissions, and even create API keys that can be used to programmatically access and manage AWS. Basically, it's your one-stop shop for all things related to user privileges, regardless of whether the "user" is a human or a script.
- **Route 53**: Although we're not going to cover Route 53 in this book, I recommend at least understanding what it is in case you need it in the future. If you do decide to utilize AWS in production, Route 53 will simplify the process of managing DNS entries and also registering new domain names. If your organization is a managed service provider, you may find yourself using this quite a bit.
- **Simple Storage Service** (**S3**): Amazon's S3 service is another offering we're not going to cover in this chapter, but it's a good idea to know that it exists and what it's for, in case you find a use for it later. S3 is actually a very popular service, and it provides **object storage**. Object storage is a new type of storage that is different than a disk (virtual or physical) that you add to your server, format, and mount. While you can still mount S3 on your server, it doesn't have a filesystem (such as ext4 or STON), nor does it understand permissions. It's simply a name-object pair, where you store files, and they have a name. With S3, you create "buckets," and each bucket can have files stored inside. Each bucket name must be unique. S3 is very useful if you want to make downloadable files available to your clients or store backup files.

- **Elastic Kubernetes** (**EKS**): In the previous chapter, we covered Kubernetes and even set up our own cluster. AWS has its own Kubernetes solution, called **EKS**. Although we're not going to cover it in this book, it's worth considering if you want to continue to use Kubernetes and have your containers running in a managed service, rather than managing your own cluster. EKS combines Kubernetes with the flexibility of the AWS platform, so it's a very useful service to consider.
- **Security Groups**: Access to many AWS resources from the public internet is disabled by default, and security groups are used to determine what is able to access a resource within AWS, and you can allow or disallow access by IP address as well as port. With regard to EC2 instances, outbound access is allowed by default, but every port is blocked inbound. You can create a security group that allows specific IPs to access the instance, which increases security. We'll see an example of this later.

Now that we have some basic understanding out of the way, we can get started and build an application in the cloud, running on AWS.

## Creating an AWS account

As mentioned in the previous section, a VPC within AWS represents a high-level abstraction of your overall network. All of the resources that we create will run inside a VPC. Therefore, we'll need to create a VPC first before we can create an EC2 instance and deploy Ubuntu.

Before we can create a VPC though, we'll need an AWS account. Before this chapter, I typically advised you to use whatever hardware you have available in order to create Ubuntu installations to work with the platform. This time, we're going to utilize an actual cloud provider, which comes at a cost. While there are free components available for a limited time with a new account, it's up to you, the reader, to keep track of billing. We'll cover costs in greater detail later in this chapter. As a general rule of thumb for now, always use whatever the cheapest option is. If a free instance type is available, go with that. Of course, if you're intending to deploy actual resources for production use within an organization, then you'll want to choose whatever instance type is appropriate for the use-case. For our purposes, we're just learning the platform for the first time, so be sure to go with the lowest-cost resources available and delete everything when you're done.

## Signing up for AWS

To get started, let's create an account. If you navigate to `https://aws.amazon.com`, you'll see an orange button on the upper-right corner with the label **Create an AWS Account**. It's possible that the layout of the page may change after publication, but you should see a button to create a new account somewhere on the page:

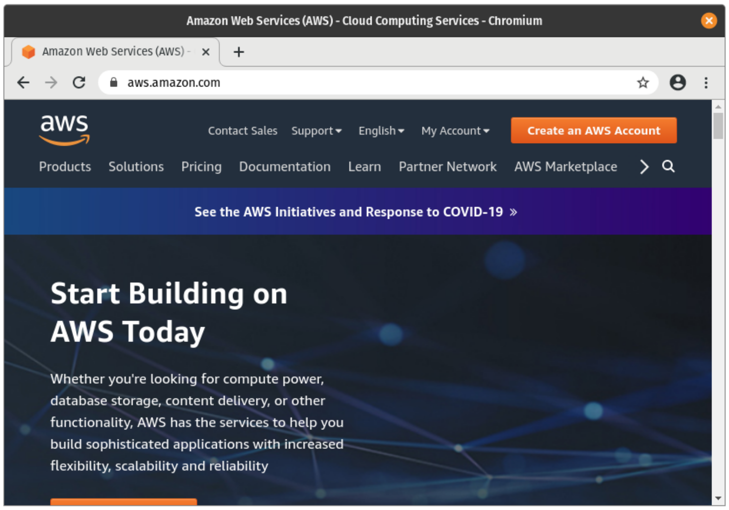
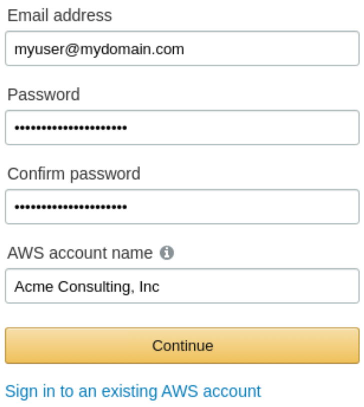

Amazon Web Services (AWS) - Cloud Computing Services - Chromium
Amazon Web Services (AWS)
aws.amazon.com
Contact Sales
Support
English
My Account
Create an AWS Account
Products
Solutions
Pricing
Documentation
Learn
Partner Network
AWS Marketplace
See the AWS Initiatives and Response to COVID-19 »
Start Building on AWS Today
Whether you're looking for compute power, database storage, content delivery, or other functionality, AWS has the services to help you build sophisticated applications with increased flexibility, scalability and reliability

Figure 19.1: The AWS main page

When you click the button to create a new account, a form will appear asking you for basic information. An example of some of the fields you may be asked for is shown in *Figure 19.2*, though the exact information you may be asked for may vary. Fill out each field accordingly, then click **Continue**:



Email address
myuser@mydomain.com

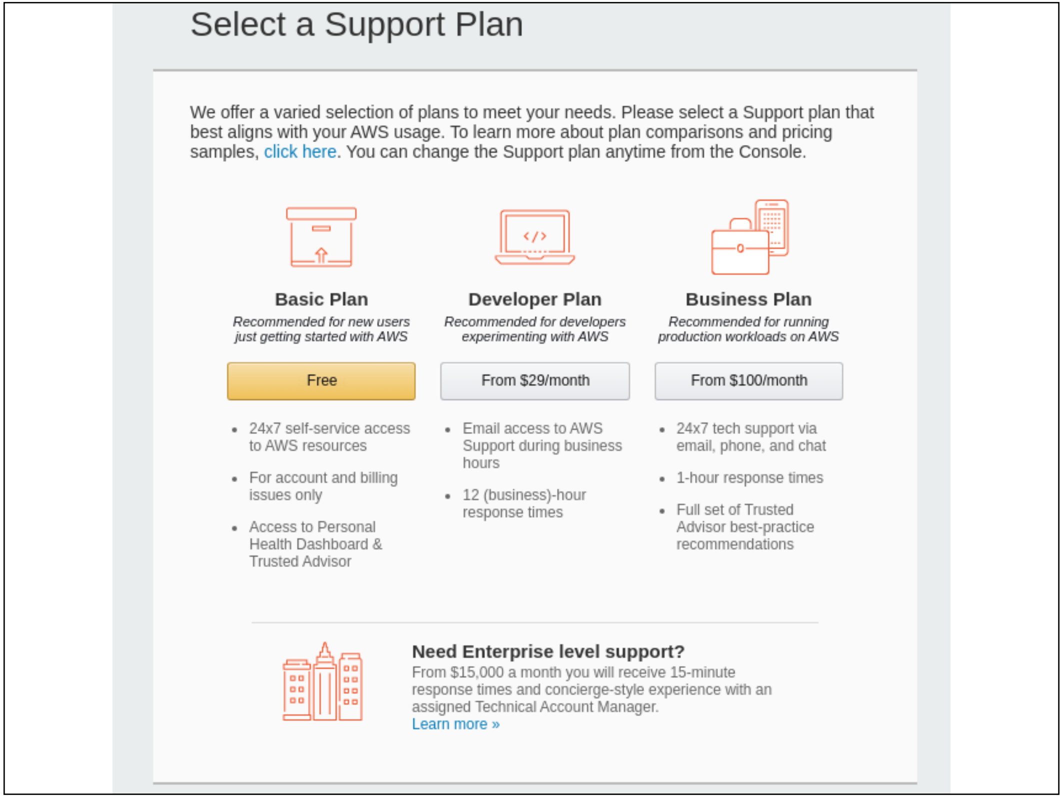
Password
••••••••••••••••••••

Confirm password
••••••••••••••••••••

AWS account name
Acme Consulting, Inc

Figure 19.2: Signing up for a new AWS account

After you click **Continue**, another screen will appear asking you to fill out additional fields, such as your full name, company name, address, and so on. This process will also include asking you to provide credit card details as well, so proceed through each screen and enter the required information. At the end of the process, you'll see a selection where you can choose your support plan:



Select a Support Plan

We offer a varied selection of plans to meet your needs. Please select a Support plan that best aligns with your AWS usage. To learn more about plan comparisons and pricing samples, click here. You can change the Support plan anytime from the Console.

**Basic Plan**
*Recommended for new users just getting started with AWS*
Free
- 24x7 self-service access to AWS resources
- For account and billing issues only
- Access to Personal Health Dashboard & Trusted Advisor

**Developer Plan**
*Recommended for developers experimenting with AWS*
From $29/month
- Email access to AWS Support during business hours
- 12 (business)-hour response times

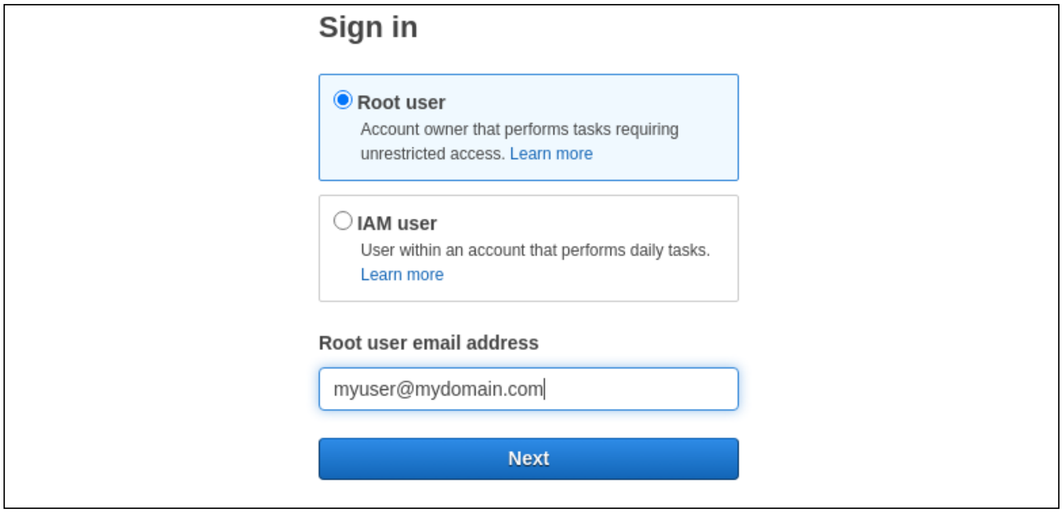
**Business Plan**
*Recommended for running production workloads on AWS*
From $100/month
- 24x7 tech support via email, phone, and chat
- 1-hour response times
- Full set of Trusted Advisor best-practice recommendations

**Need Enterprise level support?**
From $15,000 a month you will receive 15-minute response times and concierge-style experience with an assigned Technical Account Manager.
Learn more »

Figure 19.3: Choosing a support plan while signing up for an AWS account

If you are asked to select a support plan during the process, choose the one that makes sense for your use case. If you're going to be using AWS to create production instances for your organization, the Developer or Business Plan may offer additional value to you. If you're only going through the process to learn AWS and work through the examples in this chapter, choose the Basic Plan.

Once you've finished the process of creating your new account, it may take some time before your new account is ready. When it is, you should receive an e-mail saying that it's ready for use. After you do receive that e-mail, you'll be able to log in. There should be a button on the screen to sign in to the console, and then you'll see the following prompt:


Sign in

(●) Root user
Account owner that performs tasks requiring unrestricted access. Learn more

( ) IAM user
User within an account that performs daily tasks. Learn more

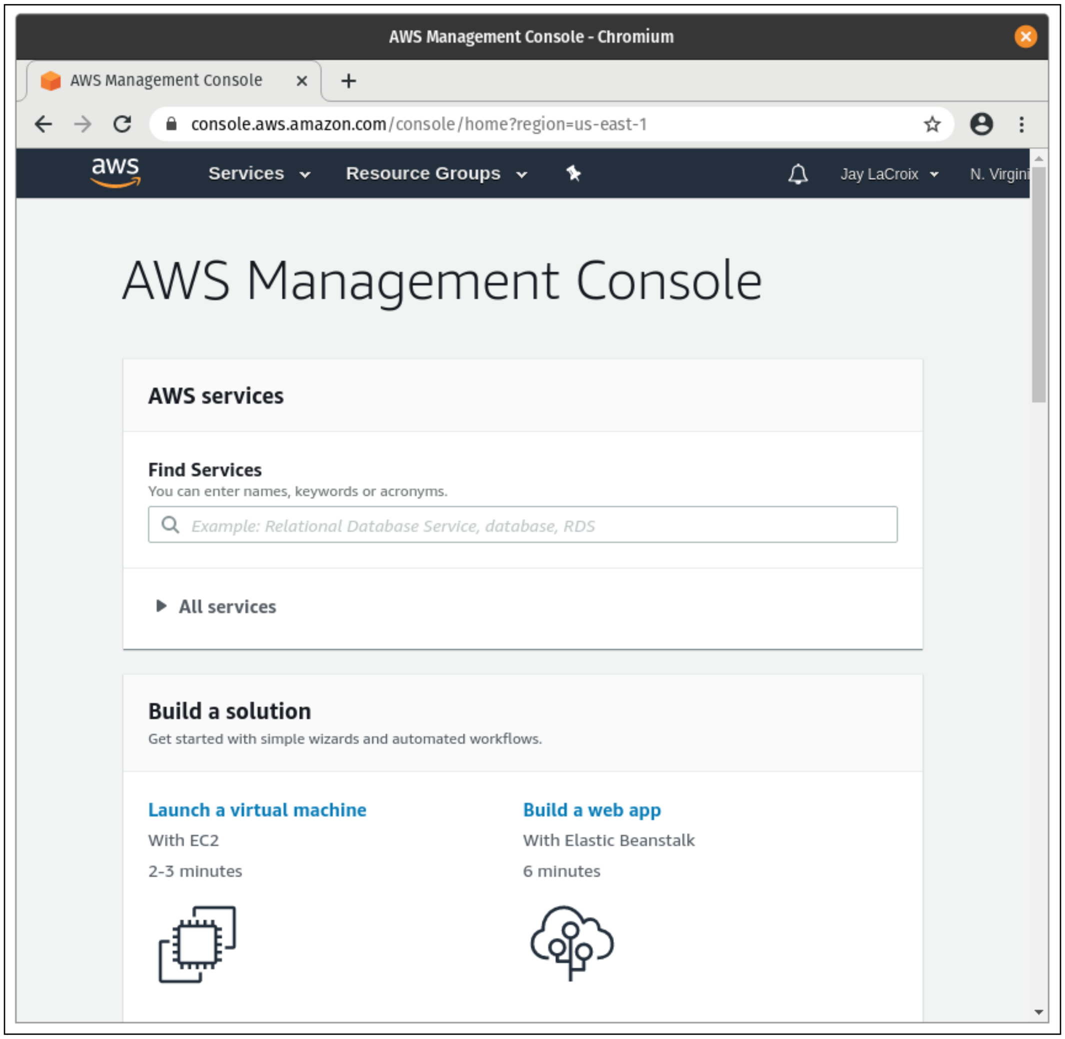
Root user email address

myuser@mydomain.com

Figure 19.4: Signing in as Root user

Here, you have a choice to log in as the root user or an IAM user. We haven't created any IAM users yet, so we only have a root user at this point. The root user is accessed by signing in with the same e-mail address that you've provided during the sign-up process. Enter that e-mail address in the prompt, and click **Next**, followed by your password on the next screen.

If all goes well, you should be logged in and see the **AWS Management Console**:


AWS Management Console - Chromium

AWS Management Console

console.aws.amazon.com/console/home?region=us-east-1

Services Resource Groups Jay LaCroix N. Virgini

AWS Management Console

AWS services

Find Services

You can enter names, keywords or acronyms.

Example: Relational Database Service, database, RDS

All services

Build a solution

Get started with simple wizards and automated workflows.

Launch a virtual machine

With EC2

2-3 minutes

Build a web app

With Elastic Beanstalk

6 minutes

Figure 19.5: The AWS Management Console main screen

Now that you have access to the management console, you can begin using AWS. Before we start creating cloud resources, we should implement some basic security to protect our account.

## Implementing basic user security

Before we continue any further, there are some very important security best practices we should employ regarding the ability to authenticate to our AWS account. Although it's very likely that our new account was created as a test account for following along with examples in this chapter, we should make it a habit to always protect our AWS account, regardless of how important it actually is.

We can begin with protecting the root account, it's a common target for hackers. Specifically, we should enable two-factor authentication for this account. Doing so will make it much harder for an outside threat to access it, since they would need access to your second factor in addition to your password. Since we're already logged in as the root user at this point, we can set this up immediately.

In the management console, you'll see a search field labeled **Find Services**. If you already know the name of the service you would like to configure, you can begin typing its name in the search field, and if your query matches an available service, it will show it in the list. If you don't know the name of the service you'd like to use, you can click on **Services** at the top-left corner of the console, to see a complete list of the services available. For setting up second-factor authentication, we will access the IAM service. You can start typing `IAM` into the search field, and it should show it as an available option. Go ahead and click on it:

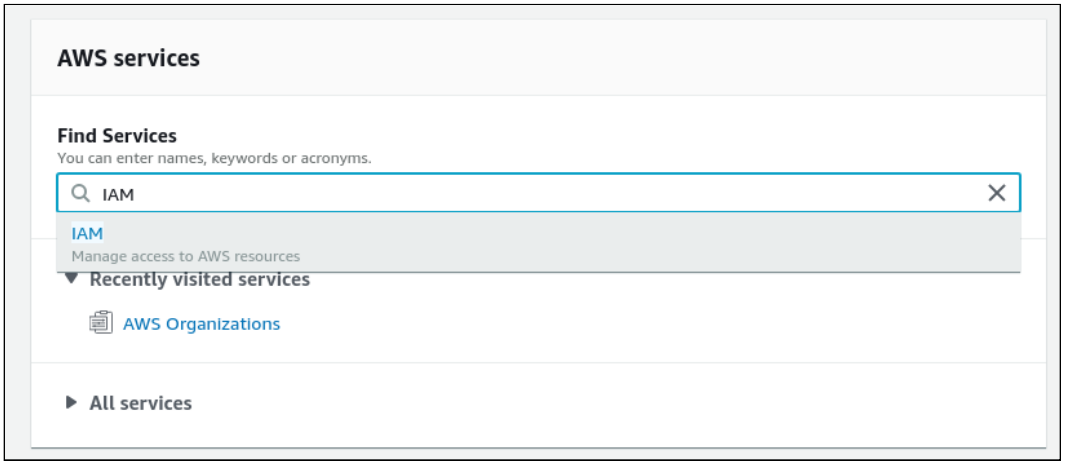
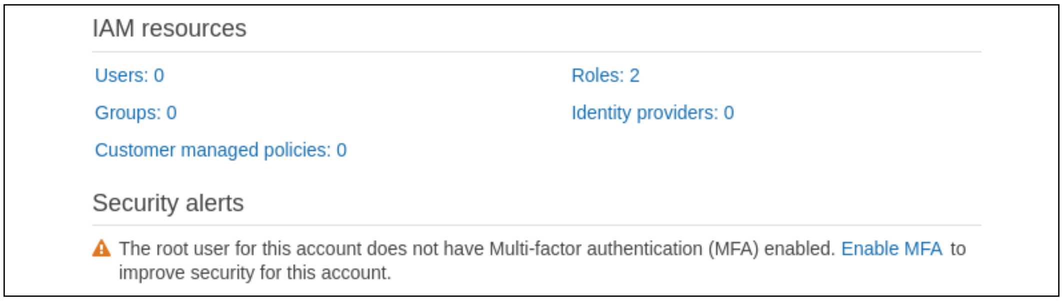

AWS services
Find Services
You can enter names, keywords or acronyms.
IAM
IAM
Manage access to AWS resources
Recently visited services
AWS Organizations
All services

Figure 19.6: Using search within the AWS Management Console to locate the IAM service

When the IAM dashboard appears, you may see a security alert that complains about the root user not having **Multi-Factor Authentication** (**MFA**) enabled. Although enabling this feature is optional, I recommend you enable it.

MFA is critical for ensuring the security of any account that is important to your organization, especially an account for a cloud computing provider such as AWS. MFA will enhance the security of the account in terms of authentication, requiring an additional factor before a user will be able to access the account. The root account is the most critical account to protect in AWS, since it has full access to every available service. To enable MFA, click on the link that reads **Enable MFA** (or similar verbiage if the layout on this page changes):


IAM resources

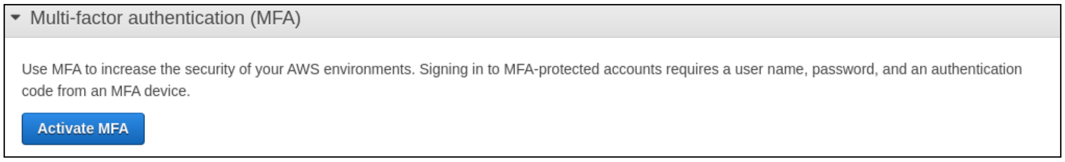
Users: 0
Roles: 2
Groups: 0
Identity providers: 0
Customer managed policies: 0

Security alerts

The root user for this account does not have Multi-factor authentication (MFA) enabled. Enable MFA to improve security for this account.

Figure 19.7: IAM dashboard, with an alert that MFA is recommended to be set up

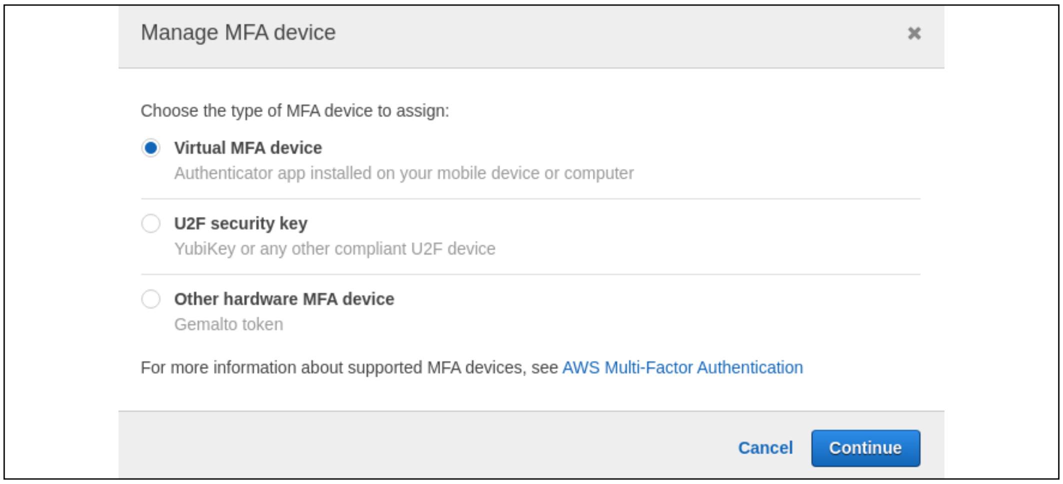
Next, click on MFA to expand it (if it isn't already), and then click on the **Activate MFA** button:


Multi-factor authentication (MFA)

Use MFA to increase the security of your AWS environments. Signing in to MFA-protected accounts requires a user name, password, and an authentication code from an MFA device.

Figure 19.8: Setting up MFA (continued)

The next screen will give you a choice of which type of device to use to facilitate MFA. The default (**Virtual MFA device**) is a good one to go with if you don't have a physical hardware key, such as a Yubikey. If you have no other preference, choose the **Virtual MFA device** option and click **Continue**:



Manage MFA device

Choose the type of MFA device to assign:

(●) **Virtual MFA device**
Authenticator app installed on your mobile device or computer

( ) **U2F security key**
YubiKey or any other compliant U2F device

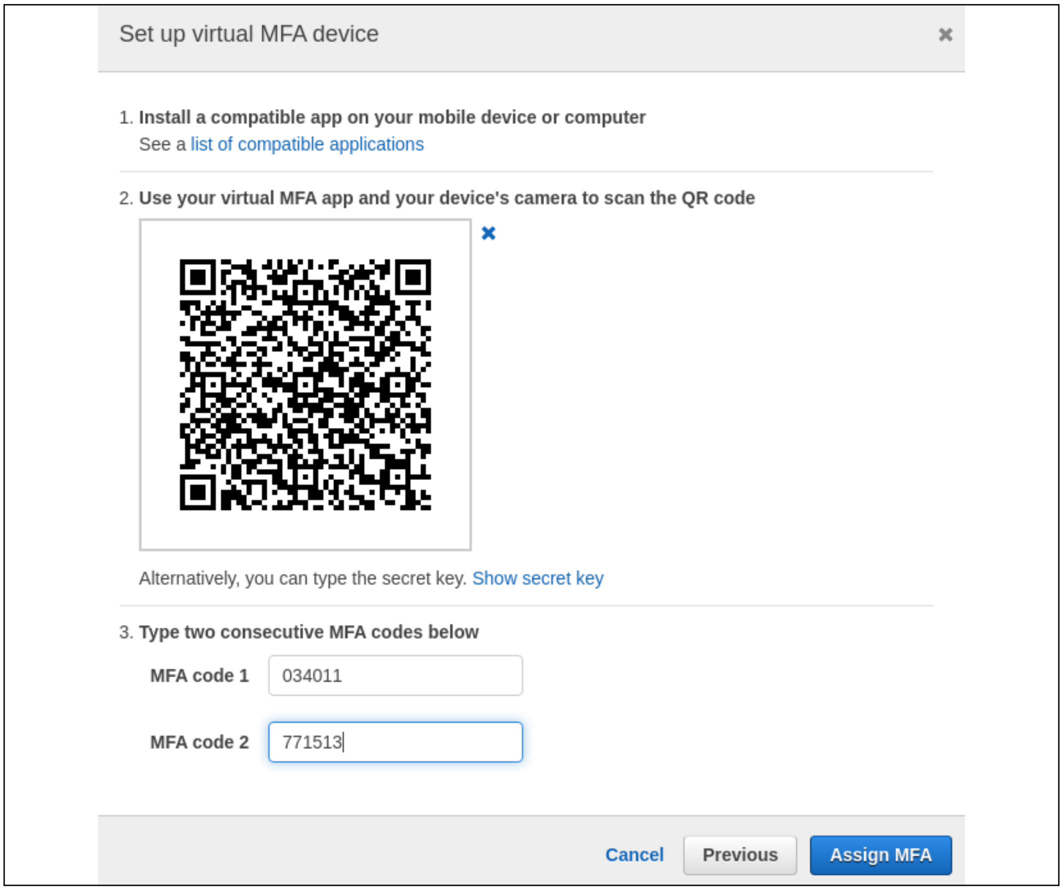
( ) **Other hardware MFA device**
Gemalto token

For more information about supported MFA devices, see AWS Multi-Factor Authentication

Figure 19.9: Choosing a type of MFA device to set up

To set up your virtual MFA device, you'll need to download a second-factor application to serve this purpose. Google Authenticator is a popular choice here, but I recommend Authy instead. Both are perfectly acceptable, but Authy also features a desktop app you can use, as well as the ability to recover your account if your primary device is not accessible for any reason. Authy is compatible with Google Authenticator, so you can typically use it with services that offer a Google Authenticator option. To continue, reveal the QR code, scan it with your phone app, and then type in two subsequent values generated by the app.

Finally, click **Assign MFA** to finalize the process:


Set up virtual MFA device

1. Install a compatible app on your mobile device or computer
   See a list of compatible applications
2. Use your virtual MFA app and your device's camera to scan the QR code

   Alternatively, you can type the secret key. Show secret key
3. Type two consecutive MFA codes below

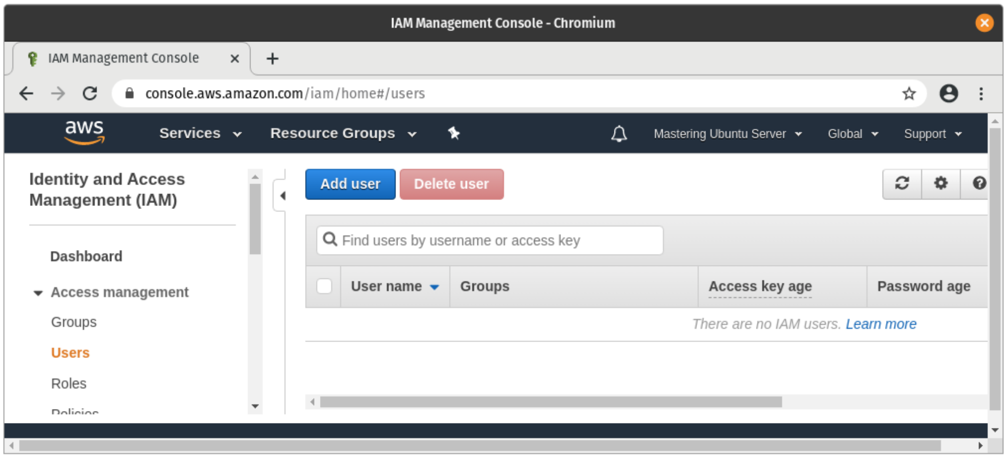
   MFA code 1: 034011

   MFA code 2: 771513

Figure 19.10: Setting up the MFA device

Now that we have two-factor authentication enabled for our root account, we should stop using that account immediately. This is actually a best practice when it comes to AWS; it's recommended that you create individual accounts for the individuals that will work on your AWS account, giving them the permissions they need to perform the tasks they need to complete. We'll discuss the **principle of least privilege** in *Chapter 21*, *Securing Your Server*, but it doesn't hurt to start thinking about the concept now. For now, what we'll do is create an administrator account for ourselves that we can use in place of the root account. We can always use the new account to create additional users if we need to do so.

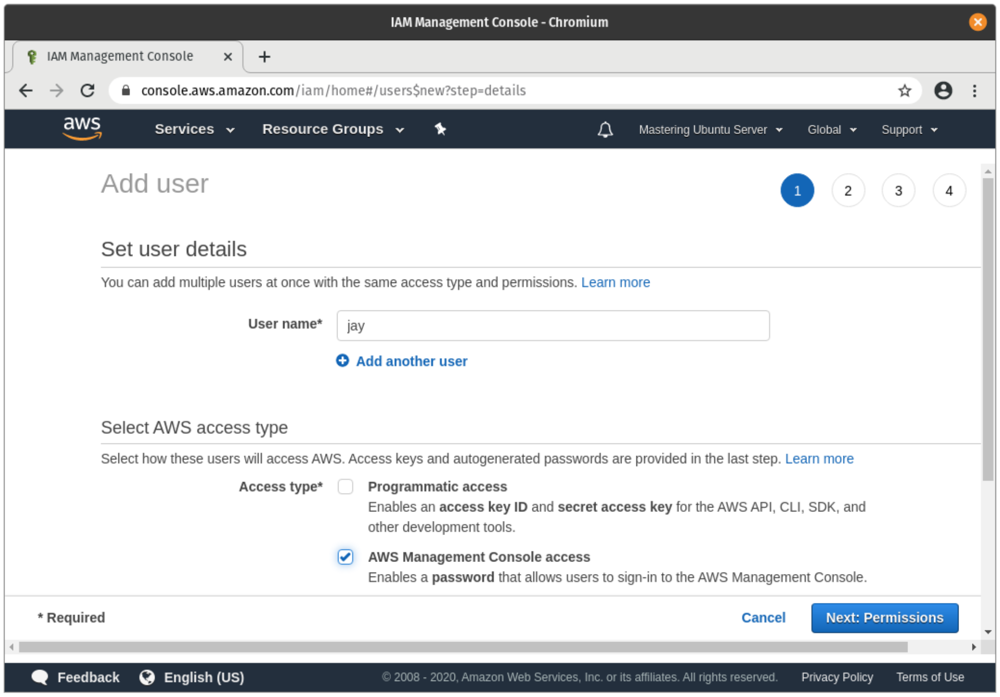
To create a new administrator account, we will again utilize the IAM console. Once there, you'll see a **Users** link on the left, and then you can click the blue **Add user** button to begin the process:



There are no IAM users. Learn more

Figure 19.11: Setting up a new administrative user for AWS

The next screen will have you type the desired username, as well as setting **Access Type**. For the username, you can name the user whatever you'd like. For **Access Type**, we'll choose **AWS Management Console access**. Click **Next: Permissions** to continue:



Add user

Set user details

You can add multiple users at once with the same access type and permissions. Learn more

User name* jay

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. Learn more

Access type* ☐ Programmatic access
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

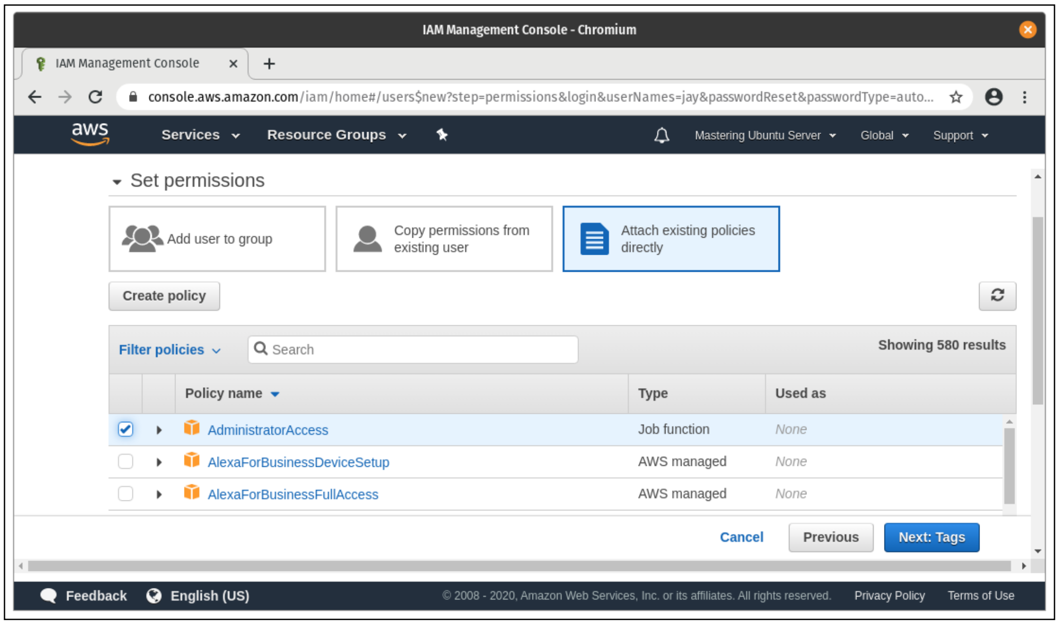
☑ AWS Management Console access
Enables a **password** that allows users to sign-in to the AWS Management Console.

* Required

© 2008 - 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Figure 19.12: Setting up a new administrative user for AWS

Next, we will set up the appropriate permissions for our user. The third icon, labeled **Attach existing policies directly**, is the first selection we'll make here, and then below that we'll place a mark next to the check box for **Administrative Access**, and then we'll click **Next: Tags**:


IAM Management Console - Chromium
IAM Management Console
console.aws.amazon.com/iam/home#/users$new?step=permissions&login&userNames=jay&passwordReset&passwordType=auto...
aws
Services
Resource Groups
Mastering Ubuntu Server
Global
Support

Set permissions

Add user to group | Copy permissions from existing user | Attach existing policies directly

Create policy

Filter policies | Search | Showing 580 results

| | | Policy name | Type | Used as |
|---|---|---|---|---|
| ✓ | ▸ | AdministratorAccess | Job function | None |
| | ▸ | AlexaForBusinessDeviceSetup | AWS managed | None |
| | ▸ | AlexaForBusinessFullAccess | AWS managed | None |

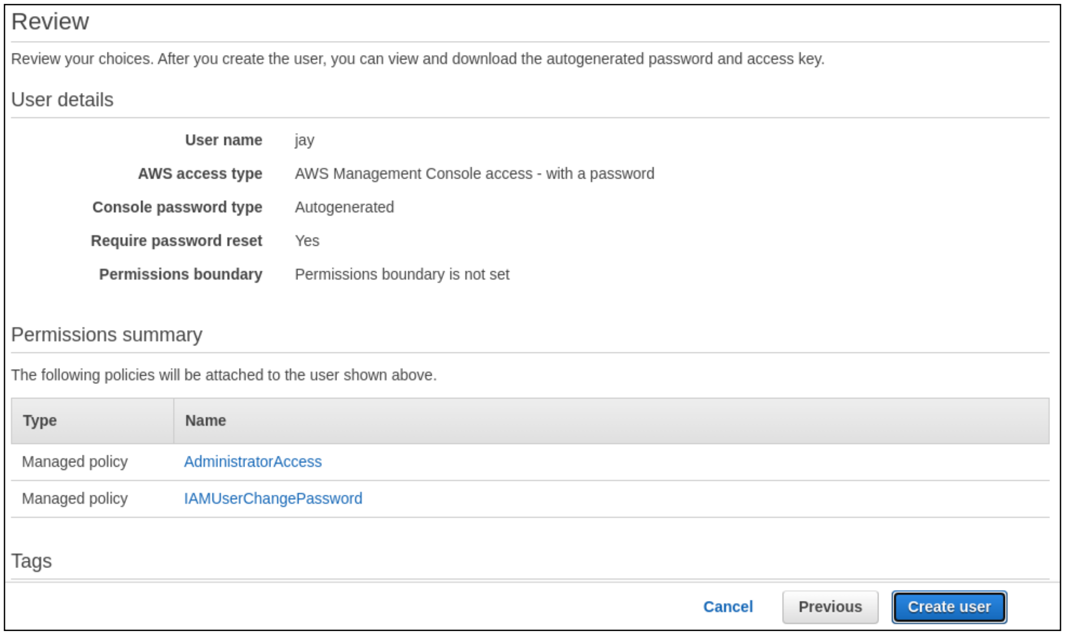
Cancel | Previous | Next: Tags

Feedback
English (US)
© 2008 - 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Figure 19.13: Setting up a new administrative user for AWS (continued)

We can skip the tags screen by leaving the fields blank and clicking **Next: Review**.

The next screen will provide us with an overview, and if everything appears to be correct here, we can click **Create User**:



Review

Review your choices. After you create the user, you can view and download the autogenerated password and access key.

User details

| | |
|---|---|
| **User name** | jay |
| **AWS access type** | AWS Management Console access - with a password |
| **Console password type** | Autogenerated |
| **Require password reset** | Yes |
| **Permissions boundary** | Permissions boundary is not set |

Permissions summary

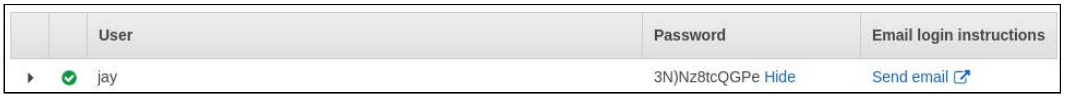
The following policies will be attached to the user shown above.

| Type | Name |
|---|---|
| Managed policy | AdministratorAccess |
| Managed policy | IAMUserChangePassword |

Tags

Figure 19.14: Setting up a new administrative user for AWS (continued)

Finally, we can retrieve the password for our new user by clicking **Show** on the next screen, which will display a randomly generated password for the user:



| User | Password | Email login instructions |
|---|---|---|
| jay | 3N)Nz8tcQGPe Hide | Send email |

Figure 19.15: Setting up a new administrative user for AWS (continued)

You can now log in to the AWS management console with the new user and the password that was provided. I recommend you use the new user going forward, as it's a good practice to not use the root account unless you absolutely have to. In addition, I recommend you follow the procedure to set up MFA as we did earlier, but this time with the new user.

## Choosing a region

As discussed earlier, a VPC within AWS is the high-level abstraction of your overall network. You can have multiple VPCs, which is similar to the concept of managing several physical networks. In fact, we already have VPCs created for us in our account, so we won't need to create one. In the future, keep in mind that creating additional VPCs is an option, should you ever need to have more than one. In our account, we have a default VPC in each **region**, so choosing which one to utilize comes down to which region is most appropriate for our use.

For production use, you'll want to create instances in AWS that are as close to your customer as you can get. For example, let's say that the customers that your organization markets to are primarily located in the Eastern United States. There's a region available within AWS that is available that's labeled **US East**, so that would be an obvious choice in that scenario. You're not limited to regions within the USA though; there are regions available all over the world, such as in Germany, China, and Canada (among others). In a nutshell, you'll want to create resources as close to your customers as possible. If you don't have a preference, you can choose to utilize whichever region is closer to you.

Although it's beyond the scope of this book, AWS offers a service called **CloudFront** that acts as a **Content Delivery Network** (**CDN**) that you can use to make your resources available in various edge locations, that users can be routed to in order to ensure they're retrieving your content from a location closest to where they are geographically. For organizations that produce media content, this is especially valuable. If this is something that might benefit you, I recommend reading more about CloudFront.

In addition, there are often multiple **availability zones** within various regions, which allow you to get even closer to your target audience. For example, when it comes to the US East region, there are two availability zones inside it, one in Virginia as well as another in Ohio. Availability zones not only give us the ability to get another step closer to our customers, they also offer us additional options for redundancy as well. For example, if one availability zone goes down for whatever reason, you can route your customers to another. Availability zones have a specific naming syntax, which consists of both the name of the region as well as the availability zone within that region. Using the Eastern United States as an example again, the two availability zones there are labeled **us-east-1** for Virginia, and **us-east-2** for Ohio. Not all regions will have multiple availability zones, though. Canada currently only has one region with one availability zone: **ca-central-1**.

In addition to availability zones, there are also **local zones**, which are intended to allow you to set up resources even closer to your customers than availability zones are able to get. Local zones are a great choice if your application is sensitive to network latency, such as running a server for an online game. We won't go over local zones in this book at all, because this is a very new offering from AWS, and there are only two of them in existence as of the time this book is being prepared for publishing. Amazon intends to add additional local zones in the future, so there may be more of them available by the time you're reading this. If your organization offers a service that is sensitive to network latency, this may be a feature you'll want to keep up to date on as they roll it out to more locations.

For now, the only consideration is which region will benefit your customers by being as close to them as possible. When it comes to following along with the examples in this book though, choosing a region closest to you geographically is a good idea.

Now that we've selected our region, how about we create an actual Ubuntu instance in the cloud? That's exactly what we'll do in the next section.

## Deploying Ubuntu as an AWS EC2 instance

With a great deal of discussion out of the way, it's time to create an actual Ubuntu deployment in the cloud. This will allow us to see the AWS service in action and give us some working experience with the EC2 service. This requires two individual steps: the first to create a required IAM role, and the second to create our instance. Let's first make sure we understand the requirements of the IAM role, then we'll set up the role and then create our new instance.

## Setting up an IAM Role for Session Manager

**Session Manager** is a service within AWS, which we can use to access a command prompt for our instance. It's actually part of **Systems Manager** and not its own service. If you want to access Session Manager, you will need to search for Systems Manager, and you'll find Session Manager as a service underneath that. You'll see this shortly.

Why should we use Session Manager? Just like with any other Linux server, we can still use OpenSSH to connect to the EC2 instance we'll be creating, just as we have many times while working with non-AWS instances throughout this book.

There's nothing wrong with using OpenSSH; with the right settings it can be a very secure option. In fact, we will explore methods of better securing it in *Chapter 21*, *Securing Your Server*. With AWS, we can use Session Manager as an alternative to OpenSSH, and it's a worthwhile alternative to learn that offers additional security in that its backend security is not something we have to manage ourselves. In addition, we can control access to it through the AWS console as well.

By default, Session Manager is not accessible at all. It requires a specific package to be installed within Ubuntu Server for it to work, and it also requires specific permissions to be enabled. The required package for Ubuntu is preinstalled by default, so the first requirement will be automatically taken care of for us immediately when we create our instance. If you have an existing Ubuntu Server in AWS that was set up before June 27th of 2018, you will need to install the required package manually:

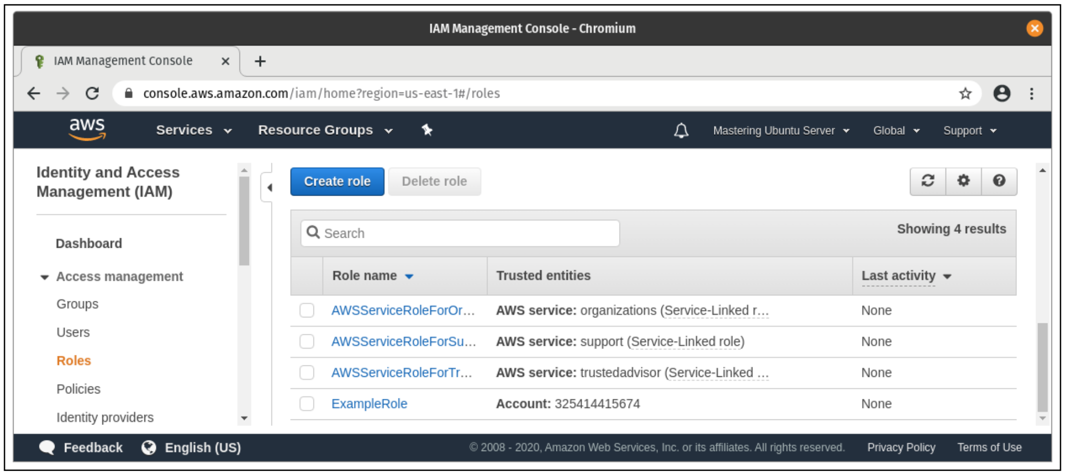
```
sudo snap install amazon-ssm-agent --classic
```

If you don't have an existing AWS account with Ubuntu Server instances created before the package was included by default, then you're all set here. As you can see from the command above though, the `amazon-ssm-agent` package is distributed as a snap package, which is a special kind of package we've discussed previously in *Chapter 3*, *Managing Software Packages*.

For the second requirement of adding permissions, we'll need to create an IAM role to allow the EC2 instance we're about to create to communicate with the Session Manager service. When I mentioned earlier that Session Manager isn't accessible by default, this is why—it's missing the permissions needed until we add them. To add the required permissions, we'll access the IAM service within the AWS console, the very same one that we used earlier to create a user account for ourselves. IAM itself has many tricks up its sleeves, more than just simply allowing us to create users. It also allows us to create **IAM roles**, which give us the ability to add permissions to entire objects. For example, we can create a role with the permissions that are required, and then we can attach that role to any EC2 instance to immediately give it the ability to be connected to by Session Manager.

Let's get started and set up the required IAM role for Session Manager. Return to the IAM section of the AWS console that we've worked with a few times now, and we'll create the required role.

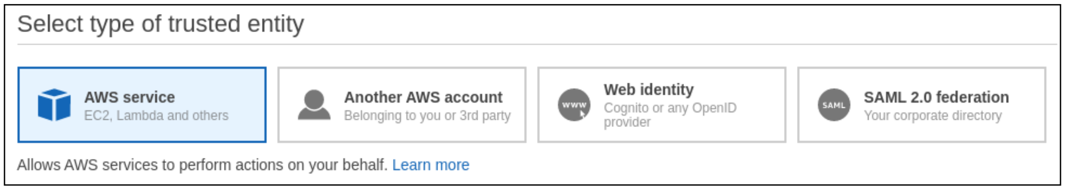
In the IAM menu on the left side of the window, click on **Roles**, and then click on the blue button labeled **Create role**:



| Role name | Trusted entities | Last activity |
|---|---|---|
| AWSServiceRoleForOr… | **AWS service:** organizations (Service-Linked r… | None |
| AWSServiceRoleForSu… | **AWS service:** support (Service-Linked role) | None |
| AWSServiceRoleForTr… | **AWS service:** trustedadvisor (Service-Linked … | None |
| ExampleRole | **Account:** 325414415674 | None |

Figure 19.16: Creating an IAM role to enable Session Manager

On the next screen, make sure **AWS service** is selected, and in the menu below that, choose **EC2** as the service. Click **Next: Permissions** to continue along:

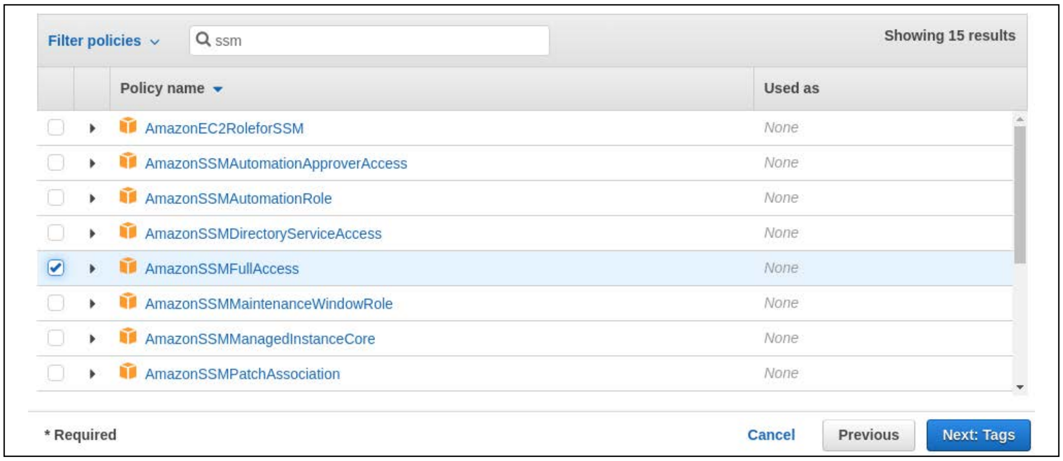


Allows AWS services to perform actions on your behalf. Learn more

Figure 19.17: Creating an IAM role to enable Session Manager (continued)

Next, we are able to attach policies to our role. In the search field next to **Filter policies**, we can type a keyword to narrow down the list, and then click on a checkbox next to a policy we wish to attach to our role. A full overview of all of the built-in policies and what they're for is beyond our scope, but as a short summary, each service within AWS has pre-built policies that can be attached to a role, which allows access to various features. Specific to our needs, we will add the **AmazonSSMFullAccess** policy to the role we're creating.

The purpose of this will become clearer when we create our EC2 instance, so for now, click **Next: Tags** to continue on:



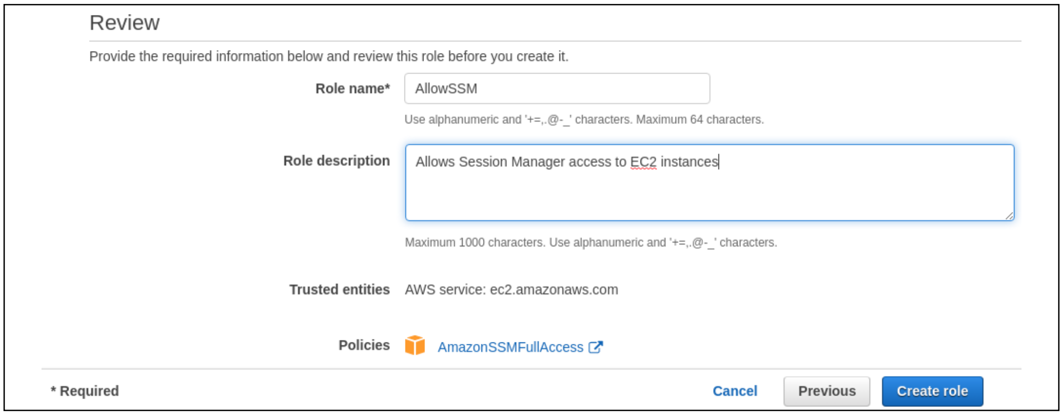
Filter policies | ssm | Showing 15 results

| | Policy name | Used as |
|---|---|---|
| ☐ | AmazonEC2RoleforSSM | None |
| ☐ | AmazonSSMAutomationApproverAccess | None |
| ☐ | AmazonSSMAutomationRole | None |
| ☐ | AmazonSSMDirectoryServiceAccess | None |
| ✓ | AmazonSSMFullAccess | None |
| ☐ | AmazonSSMMaintenanceWindowRole | None |
| ☐ | AmazonSSMManagedInstanceCore | None |
| ☐ | AmazonSSMPatchAssociation | None |

Figure 19.18: Attaching the AmazonSSMFullAccess policy to our role

The next screen that will appear will give us the ability to add one or more tags. We'll skip this for now, but you can feel free to add any tags you'd like here. Tags allow you to attach information to a resource and aren't limited to IAM roles. You can add any descriptive information you feel is pertinent, if you wish. Tags are simply "key: value" pairs, so there's no specific naming scheme to follow here. Add tags if you wish to do so, and when you're finished with this screen click **Next: Review**.

The final screen will give us a review of the settings we've chosen so far, as well as an option to name the role, and add a description if we wish to do so. Although it's optional, I recommend giving the role a name, to make it easier to identify later. When you're finished, click **Create role**:



Review

Provide the required information below and review this role before you create it.

Role name*: AllowSSM

Use alphanumeric and '+=,.@-_' characters. Maximum 64 characters.

Role description: Allows Session Manager access to EC2 instances

Maximum 1000 characters. Use alphanumeric and '+=,.@-_' characters.

Trusted entities: AWS service: ec2.amazonaws.com

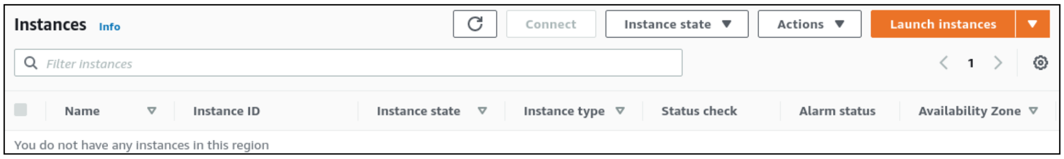
Policies: AmazonSSMFullAccess

Figure 19.19: Adding a name and description to our role

When it comes to setting up our IAM role, we're all set—the role has been created and we can go ahead and use it. Next, it's time to create our Ubuntu instance.

## Creating an Ubuntu Server instance in AWS

Now it's time to see our work come together and create our Ubuntu instance. In the AWS console, we should first access the EC2 service to get started. You can easily find any service by typing it into the search box within the console, so if you start typing EC2 into that field, you should see **EC2** on the list. Alternatively, you can click on **Services** in the top-left corner of the console window. After that, click on **Instances** on the left side of the screen. After doing that, you'll see a screen with a button labeled **Launch instances**:

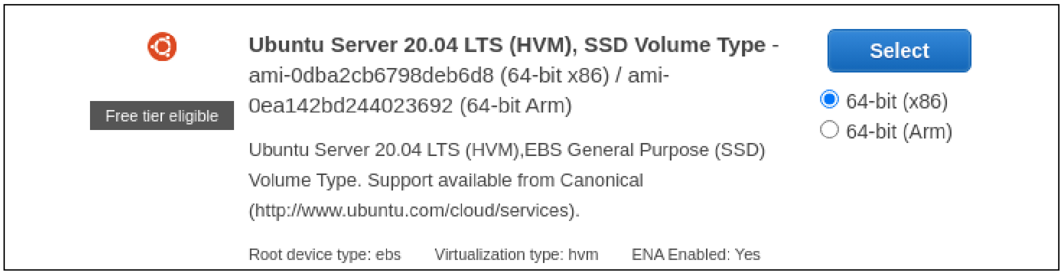


| | Name | Instance ID | Instance state | Instance type | Status check | Alarm status | Availability Zone |
|---|---|---|---|---|---|---|---|

You do not have any instances in this region

Figure 19.20: The main window of the EC2 service

Normally, the **Instances** section of the EC2 console will show us a list of all of our server instances, but unless you've read ahead, we don't have any yet so the window is blank. When you click on **Launch instances**, you'll see various operating systems on the list, and there are quite a few. If you start typing `Ubuntu` in the search field, you should see an option for Ubuntu 20.04, similar to *Figure 19.21*:



Ubuntu Server 20.04 LTS (HVM), SSD Volume Type - ami-0dba2cb6798deb6d8 (64-bit x86) / ami-0ea142bd244023692 (64-bit Arm)

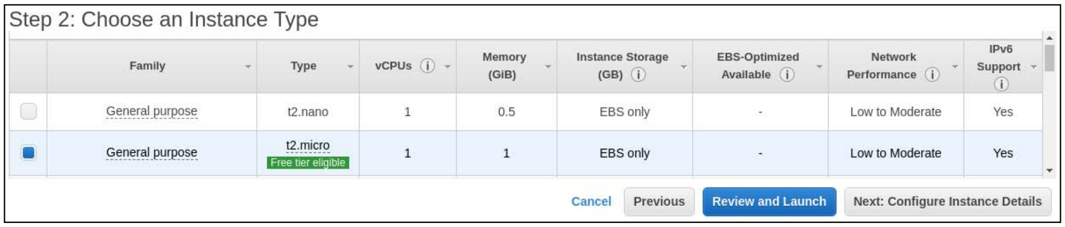
Ubuntu Server 20.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (http://www.ubuntu.com/cloud/services).

Root device type: ebs  Virtualization type: hvm  ENA Enabled: Yes

[✓ 64-bit (x86)] 64-bit (Arm)

Figure 19.21: The Ubuntu option for EC2

Click on the **Select** button to proceed. Once you do, you should see a list of instance types similar to *Figure 19.22*:



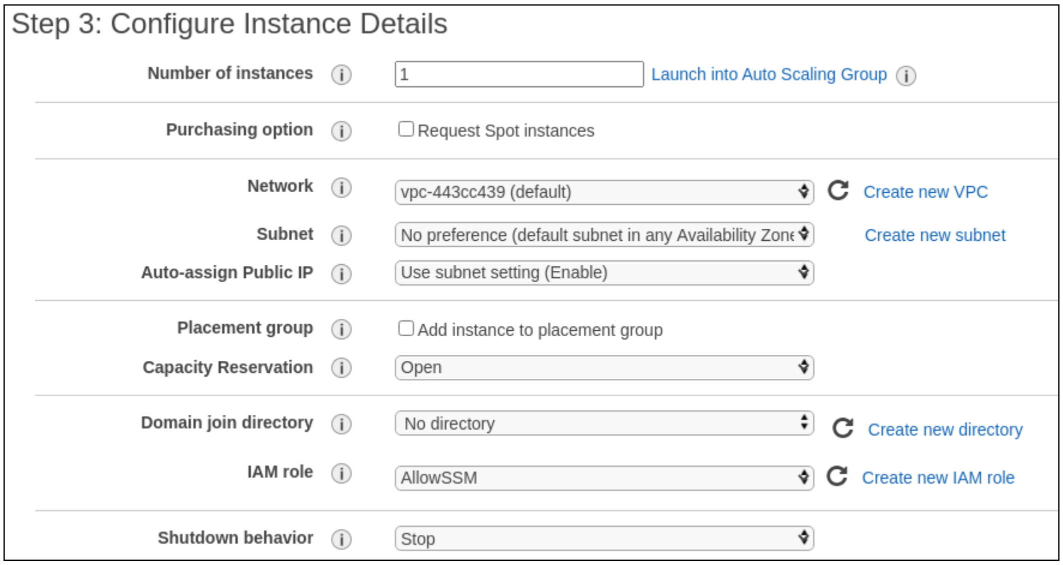
Step 2: Choose an Instance Type

| | Family | Type | vCPUs | Memory (GiB) | Instance Storage (GB) | EBS-Optimized Available | Network Performance | IPv6 Support |
|---|---|---|---|---|---|---|---|---|
| ☐ | General purpose | t2.nano | 1 | 0.5 | EBS only | - | Low to Moderate | Yes |
| ■ | General purpose | t2.micro Free tier eligible | 1 | 1 | EBS only | - | Low to Moderate | Yes |

Figure 19.22: Choosing an instance type

On the **Choose an Instance Type** screen, there will be quite a few instance types from which to choose. I've shrunk *Figure 19.22* down to show the first two options, but there are many more. I selected the **t2.micro** instance type, and I recommend you do the same. It's eligible for the free tier, which is a special tier you'll have access to within the first 12 months of the age of the account. You most likely wouldn't choose this instance type for a production server, as it will be quite slow—it only has 1 CPU and 1GiB of memory. And it's a burstable instance type, which means the speed fluctuates based on usage. It's able to burst to take care of busy workloads, but its ability to do so depends on CPU credits that it earns in a particular time. A full explanation is beyond the scope of this chapter, but if you're going to use AWS in production, it's a good idea to read about the various instance types available. Although you don't see it in the screenshot, you should also see the cost on this page as well. But again, we'll utilize the free tier for now. Choose **t2.micro** as the instance type, then click **Next: Configure Instance Details**.

The next screen that appears will allow us to choose individual details about our instance, and we'll have quite a few options to choose from. We'll accept the defaults for most of these options:



Step 3: Configure Instance Details

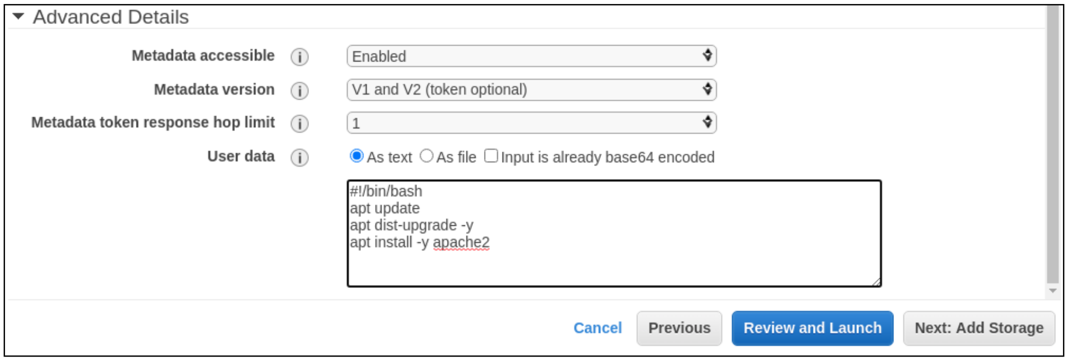
| | |
|---|---|
| Number of instances | 1 |
| Purchasing option | ☐ Request Spot instances |
| Network | vpc-443cc439 (default) |
| Subnet | No preference (default subnet in any Availability Zone |
| Auto-assign Public IP | Use subnet setting (Enable) |
| Placement group | ☐ Add instance to placement group |
| Capacity Reservation | Open |
| Domain join directory | No directory |
| IAM role | AllowSSM |
| Shutdown behavior | Stop |

Figure 19.23: Setting options for our new EC2 instance

The first field gives us the ability to create more than one server, otherwise by default we will create a single server each time we launch an instance. If you do decide to create multiple instances, remember to keep track of them and delete them when you're done, so you don't get an actual charge on your bill.

We can also choose the **Network** and **Subnet** for our instance as well, but we'll leave the defaults for that. With advanced usage of AWS, you can create multiple public and private networks and have the EC2 instance utilize specific ones. We won't work on configuring that in this chapter, but keep in mind that it's a possibility to do so.

Skipping ahead a bit, I did change the **IAM role** option, and I set it to the IAM role that we created earlier. The AllowSSM role that we set up provides access to Session Manager, so by attaching this role to our instance, we're ensuring that we'll be able to use Session Manager to connect to it. If we omit this role, then connections to the instance via that service will fail. We'll see Session Manager in action shortly. But for now, scroll down a bit further as there is an additional option we'll want to set up:



Metadata accessible: Enabled
Metadata version: V1 and V2 (token optional)
Metadata token response hop limit: 1
User data: [✓ As text] As file ☐ Input is already base64 encoded

```
#!/bin/bash
apt update
apt dist-upgrade -y
apt install -y apache2
```

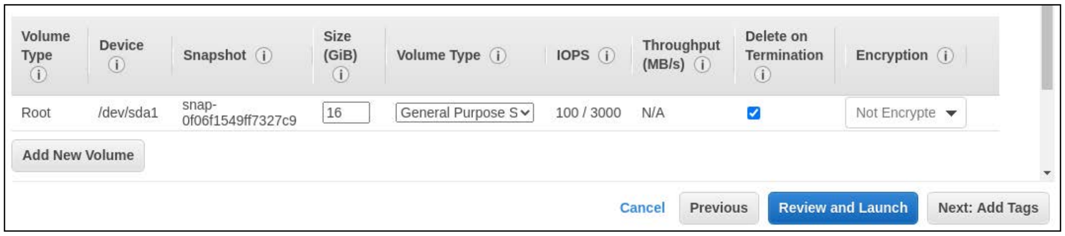
Figure 19.24: Adding user data for the EC2 instance

At the bottom of the same screen, we see an option to add **User data**. Sometimes overlooked, user data is a very powerful feature of AWS. The name can be misleading, we're not adding information about a user account here. What it's actually used for is adding commands that we want to run when the instance is being created. I've added the following code to the **User data** field in my case:

```
#!/bin/bash
apt update
apt dist-upgrade -y
apt install -y apache2
```

If you think the code resembles a Bash script, then you're correct—it is. I added four lines of Bash statements to the **User data** field, to have some commands run automatically. The code should be relatively straightforward: I have it set up to update the repository index, then perform a full system upgrade.

This is important; we always want our servers to start with the latest patches available. As a proof of concept, I added a statement to install Apache. Notice that I included the `-y` option to all of the `apt` commands. This automatically responds "yes" to any question `apt` may ask, since we don't have a display hooked up to this server and are unable to answer questions ourselves. Without that option, the user data will fail to apply. Anyway, click **Next: Add Storage** to continue. On the next screen that appears, we'll set some parameters regarding storage:

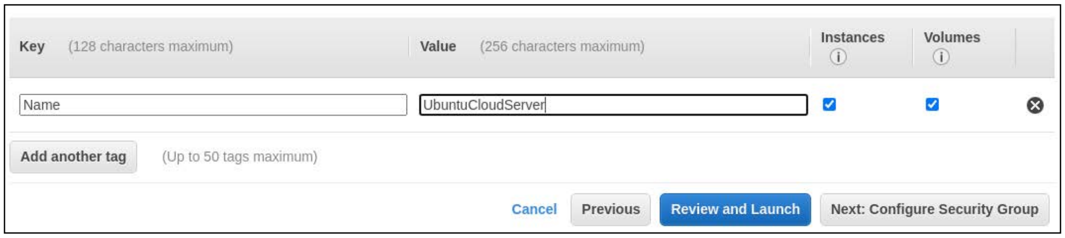


| Volume Type | Device | Snapshot | Size (GiB) | Volume Type | IOPS | Throughput (MB/s) | Delete on Termination | Encryption |
|---|---|---|---|---|---|---|---|---|
| Root | /dev/sda1 | snap-0f06f1549ff7327c9 | 16 | General Purpose S | 100 / 3000 | N/A | ✓ | Not Encrypte |

Figure 19.25: Adding user data for the EC2 instance

The most important decision on this screen is how much storage to provide our EC2 instance. I chose 16 GiB in my case. You'll want to make sure that this is large enough to accommodate whatever it is you intend to use the server for. You can also click **Add New Volume**, to add secondary storage. If you do, then you'll need to format and mount this storage manually after the instance comes online, similar to how we added additional storage volumes back in *Chapter 9*, *Managing Storage Volumes*. Refer back to that chapter if you do decide to add additional storage and need a refresher on how to format the volume and get it mounted. Go ahead and click **Next: Add Tags** to continue.

The next course of action will be to add tags, if we wish to do so:

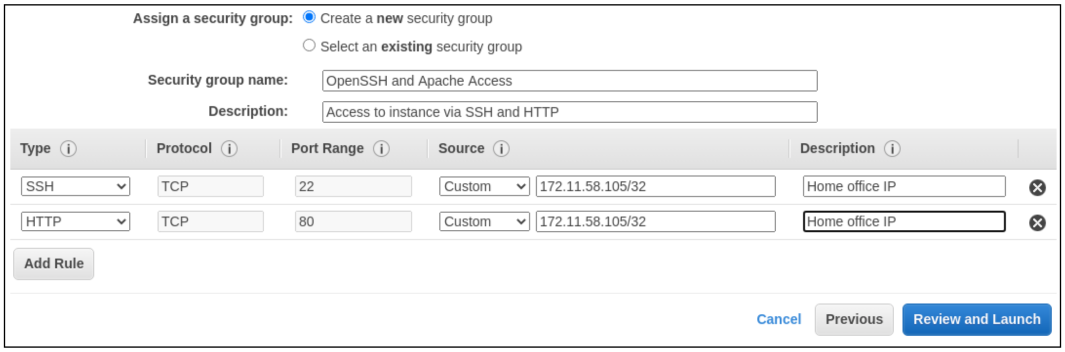


| Key (128 characters maximum) | Value (256 characters maximum) | Instances | Volumes |
|---|---|---|---|
| Name | UbuntuCloudServer | ✓ | ✓ |

Figure 19.26: Adding tags to our instance

Adding tags is optional, but I recommend you at least add a tag called **Name** and set the value to whatever you would like to call your instance. This makes it easier to find in the list and is a good habit to get into.

There are no specific requirements here, so continue on by clicking **Next: Configure Security Group**:



Assign a security group: ● Create a **new** security group ○ Select an **existing** security group

Security group name: OpenSSH and Apache Access

Description: Access to instance via SSH and HTTP

| Type | Protocol | Port Range | Source | | Description |
|---|---|---|---|---|---|
| SSH | TCP | 22 | Custom | 172.11.58.105/32 | Home office IP |
| HTTP | TCP | 80 | Custom | 172.11.58.105/32 | Home office IP |

Figure 19.27: Configuring a security group for the instance

**Security groups**, as briefly touched on earlier in this chapter, give you the ability to allow additional communication to occur to the instance. By default, outbound is completely enabled. Anything from within the instance attempting to access the outside internet will be allowed. The reverse isn't allowed though; nothing is allowed to access the instance from the outside until you allow it.

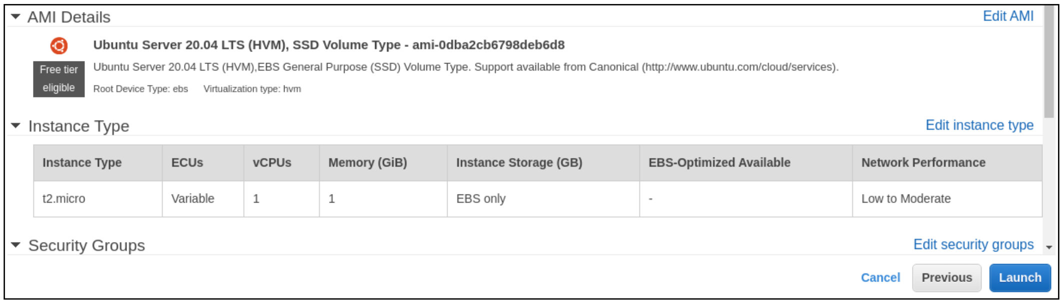
Let's spend some time and carefully consider what we allow here, because security groups are a critical consideration. In the example screenshot, I've added a random IP twice, the first for accessing SSH and the second for Apache. Standard practice would be to add the external IP address for your internet connection here, and only requests coming from that IP are allowed to communicate to the instance via that port.

If your instance will serve a website to the public internet, then the IP address you would add is `0.0.0.0/0`. If you allow a port to be accessed by 0.0.0.0/0, then that means you're allowing every IP address to be able to connect to that port. In the case of hosting a public website, then that is likely what you'll want. But the general rule of thumb is to only allow specific IP addresses to connect to a port on your instance, unless you have a very important and unavoidable reason to allow public access to it.

When it comes to port `22` for OpenSSH, you should never allow public access to that. In the screenshot above, let's assume `172.11.58.105` is the IP address for my home office. I added `/32` to the end of that IP address, which signifies that it's not referring to a network but instead a specific IP address. It's standard practice to allow the IP address of your organization and/or your home office to access OpenSSH, but nothing else should be able to do so. This keeps OpenSSH away from hackers and outside threats but still gives you the ability to allow public connections to something like Apache, if you want to do so.

In my case, I limited access to Apache as well as OpenSSH. It's a good idea to be very restrictive when it comes to allowing things from the outside to access your server. Always default to "no" unless you have no other choice.

With the security group created, the next screen will give you an overview of our selections so far, and if everything looks appropriate, click **Launch**:



Ubuntu Server 20.04 LTS (HVM), SSD Volume Type - ami-0dba2cb6798deb6d8

Free tier eligible

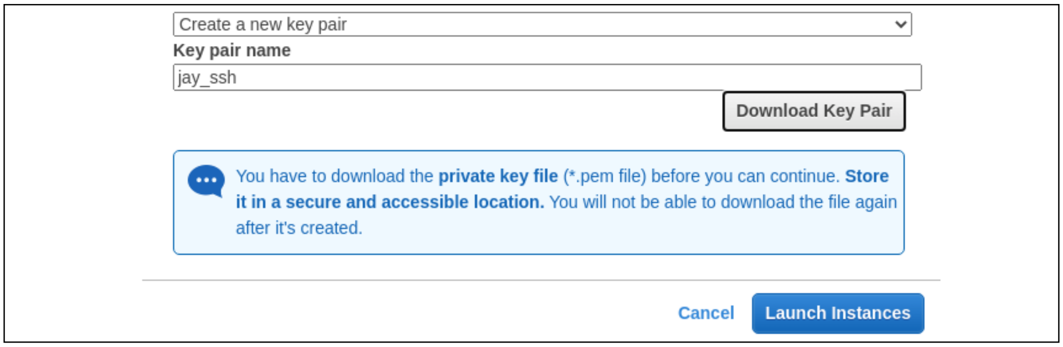
Ubuntu Server 20.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (http://www.ubuntu.com/cloud/services).

Root Device Type: ebs    Virtualization type: hvm

| Instance Type | ECUs | vCPUs | Memory (GiB) | Instance Storage (GB) | EBS-Optimized Available | Network Performance |
|---|---|---|---|---|---|---|
| t2.micro | Variable | 1 | 1 | EBS only | - | Low to Moderate |

Figure 19.28: Preparing to launch an instance

Although the button in the previous step was labeled **Launch**, the instance won't launch yet—a very important screen will now appear. At this stage, you'll create an SSH key pair to access your instance. We'll be using Session Manager to connect to it, but it's a good idea to create a key here. In fact, you can't continue until you choose to create a new key and download it or select an existing key. Since we haven't created a key yet, type a name for the key pair and click on the **Download Key Pair** button. Be sure to keep the key in a safe place: you'll literally never be able to download it again, this is your only chance to do so. If you lose it, you cannot retrieve it. After you've downloaded the key, click **Launch Instances** to continue:


Create a new key pair

Key pair name: jay_ssh

You have to download the **private key file** (*.pem file) before you can continue. **Store it in a secure and accessible location.** You will not be able to download the file again after it's created.

Figure 19.29: Creating an OpenSSH key pair for the new instance

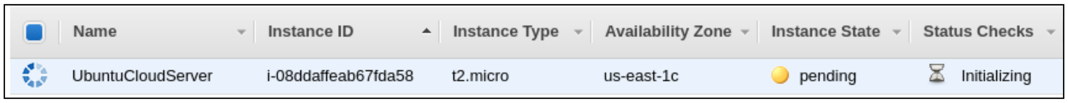
With regards to the key file that you'll end up downloading, you can use it to connect to your instance via OpenSSH with a command similar to the following:

```
ssh -i /path/to/key.pem ubuntu@<Instance Public IP>
```

For me, if I add the path of my key as well as the public IP address listed for my instance, the command becomes this:

```
ssh -i /home/jay/downloads/jay_ssh.pem ubuntu@54.81.234.225
```

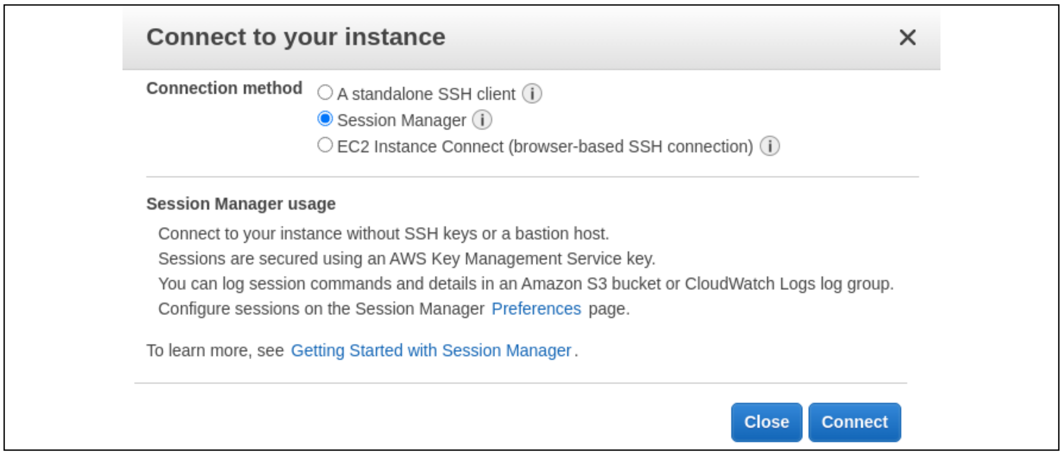
Again, we'll use Session Manager shortly, but it's nice to have another method of accessing our instance if Session Manager ever fails us for some reason. At this point, we should see the new instance in the list of EC2 instances in our account, and it will take some time for it to be ready for use:



| | Name | Instance ID | Instance Type | Availability Zone | Instance State | Status Checks |
|---|---|---|---|---|---|---|
| | UbuntuCloudServer | i-08ddaffeab67fda58 | t2.micro | us-east-1c | ● pending | Initializing |

Figure 19.30: The new EC2 instance listed, and in the process of being provisioned

In *Figure 19.30*, the **Instance State** field shows a status of **pending**, which means that the instance is being prepared and after 5-10 minutes or so, it will change to **available**. Once it does, we will be able to connect to our instance. To do so, right-click on the instance and click **Connect**, and the following window should appear:



Connect to your instance

Connection method: A standalone SSH client [✓ Session Manager] EC2 Instance Connect (browser-based SSH connection)

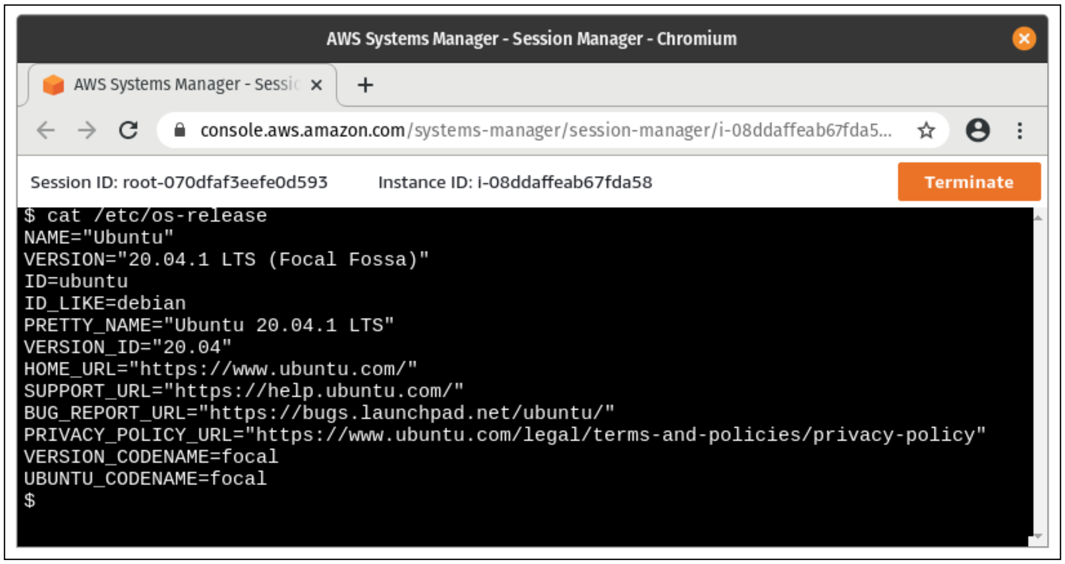
Session Manager usage

Connect to your instance without SSH keys or a bastion host.
Sessions are secured using an AWS Key Management Service key.
You can log session commands and details in an Amazon S3 bucket or CloudWatch Logs log group.
Configure sessions on the Session Manager Preferences page.

To learn more, see Getting Started with Session Manager.

Figure 19.31: Choosing a method to use to connect to the EC2 instance

As you can see in *Figure 19.31*, I chose **Session Manager** as the connection method, and I recommend you do the same. Then, click **Connect** to attempt to initiate a console window with a connection to our instance. If successful, you should see an actual command prompt just as you would if you used OpenSSH:


AWS Systems Manager - Session Manager - Chromium
AWS Systems Manager - Sessic
console.aws.amazon.com/systems-manager/session-manager/i-08ddaffeab67fda5...
Session ID: root-070dfaf3eefe0d593
Instance ID: i-08ddaffeab67fda58
Terminate

```
$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.1 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.1 LTS"
VERSION_ID="20.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=focal
UBUNTU_CODENAME=focal
$
```

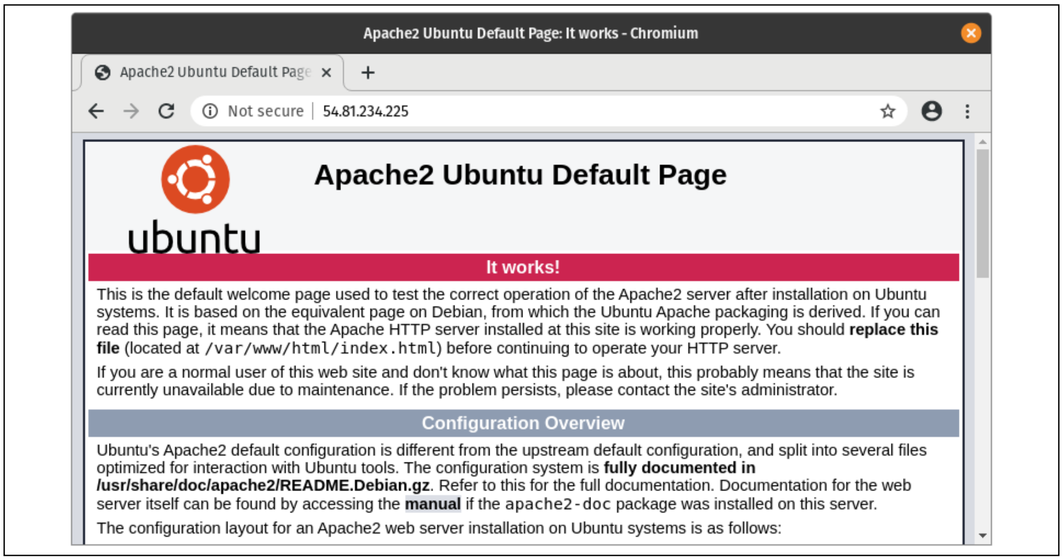
Figure 19.32: An active Session Manager connection to the new EC2 instance

In *Figure 19.32*, I entered the following command in order to display details for the version of Ubuntu that was deployed in the instance:

```
cat /etc/os-release
```

The `/etc/os-release` file is included with all Ubuntu installations, and as you can see from the output, it contains some information regarding the version of Ubuntu we're running. That command was entered directly into the Session Manager window, to show that the connection is actually working and we can now configure the instance right from within our web browser!

If you recall, we added User Data earlier, and that included a command to install Apache. If you enter the public IP address of your EC2 instance into a web browser, you should see the default Apache web page:


Apache2 Ubuntu Default Page: It works - Chromium
Apache2 Ubuntu Default Page
Not secure | 54.81.234.225
ubuntu

Apache2 Ubuntu Default Page

It works!

This is the default welcome page used to test the correct operation of the Apache2 server after installation on Ubuntu systems. It is based on the equivalent page on Debian, from which the Ubuntu Apache packaging is derived. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should **replace this file** (located at `/var/www/html/index.html`) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

Configuration Overview

Ubuntu's Apache2 default configuration is different from the upstream default configuration, and split into several files optimized for interaction with Ubuntu tools. The configuration system is **fully documented in /usr/share/doc/apache2/README.Debian.gz**. Refer to this for the full documentation. Documentation for the web server itself can be found by accessing the **manual** if the `apache2-doc` package was installed on this server.

The configuration layout for an Apache2 web server installation on Ubuntu systems is as follows:

Figure 19.33: The Apache default web page running on an EC2 instance

Congratulations! You've successfully deployed Ubuntu Server to the cloud, and now have an actual web server running on it. That's all there is to it. Using Session Manager is also simple; all you need to do to customize the server further is right-click on it, click **Connect**, and you can then continue to build the instance. That's awesome!

What's not so awesome, though, is when something happens to your server and you have to start over and rebuild it from scratch. In the next section, we're going to explore the process of creating an image of the server that we can utilize to deploy customized versions of Ubuntu.

## Creating and deploying Ubuntu AMI images

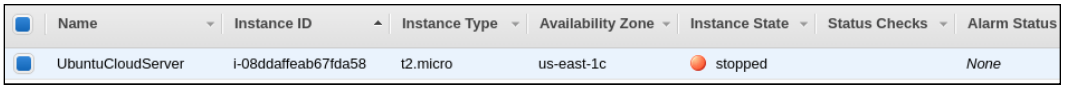
Just about every cloud platform I know of includes some sort of feature that can be used to create images of the instance's hard disk. An image can be used to create copies of the original server, as well as acting as a starting point so if the server needs to be rebuilt, we won't have to start over from scratch. In AWS, images are known as **Amazon Machine Images** (**AMIs**). For all intents and purposes, there's nothing very unique about AMIs; if you've worked with disk images in the past, it's the same thing. When it comes to what you should include in an AMI, you can (and should) use your imagination here—anything you find yourself manually setting up or configuring while rolling out a new server is a candidate to be included in an image, and the more customizations you include inside the image, the more time it will save you later.

Let's see this in action and create an image of the server we've just set up. We should consider shutting down our server first, although this isn't required. Taking an AMI of a server that is shut down is preferred over doing the same on a server that is running. When the server is shut down, nothing is writing to its disk, so we don't have to worry about corruption if we're capturing an AMI in the middle of a critical operation. The likelihood of running into an issue while creating an AMI of a running server is very small, but I recommend shutting down the server if you can to be on the safe side.

In the EC2 console of AWS, you can right-click on the instance to access **Session Manager**, and then you can simply power it off from the command prompt:

```
sudo poweroff
```

In AWS, it can take a minute or two for an instance to power down. You can refresh the page after some time, and the status should change to **stopped**:



| | Name | Instance ID | Instance Type | Availability Zone | Instance State | Status Checks | Alarm Status |
|---|---|---|---|---|---|---|---|
| | UbuntuCloudServer | i-08ddaffeab67fda58 | t2.micro | us-east-1c | ● stopped | | *None* |

Figure 19.34: The Apache default web page running on an EC2 instance

Once the instance has stopped, you can right-click on it to begin the process of creating an AMI. Hover over **Images and Templates** and then click **Create Image**:

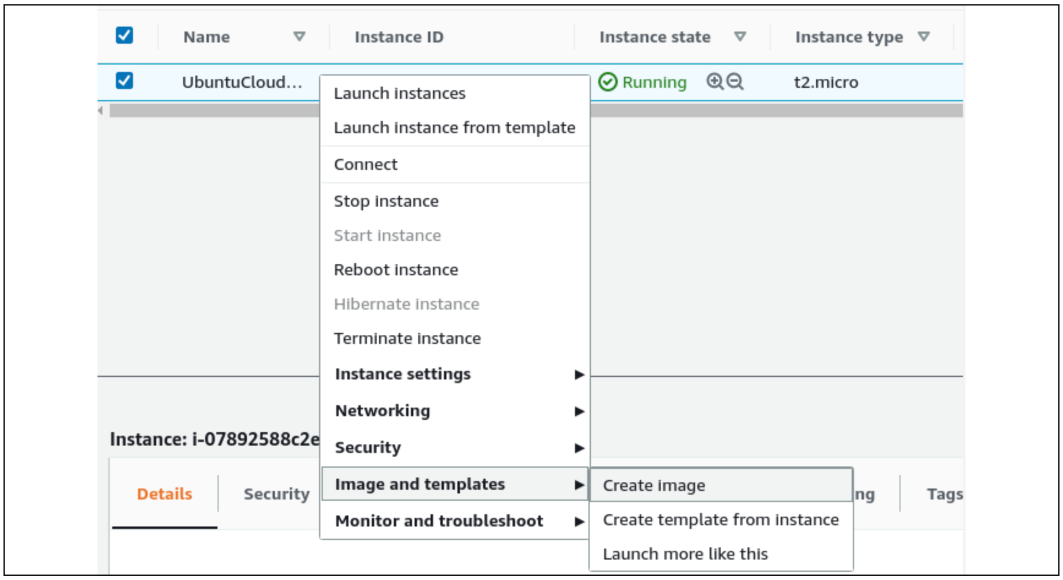
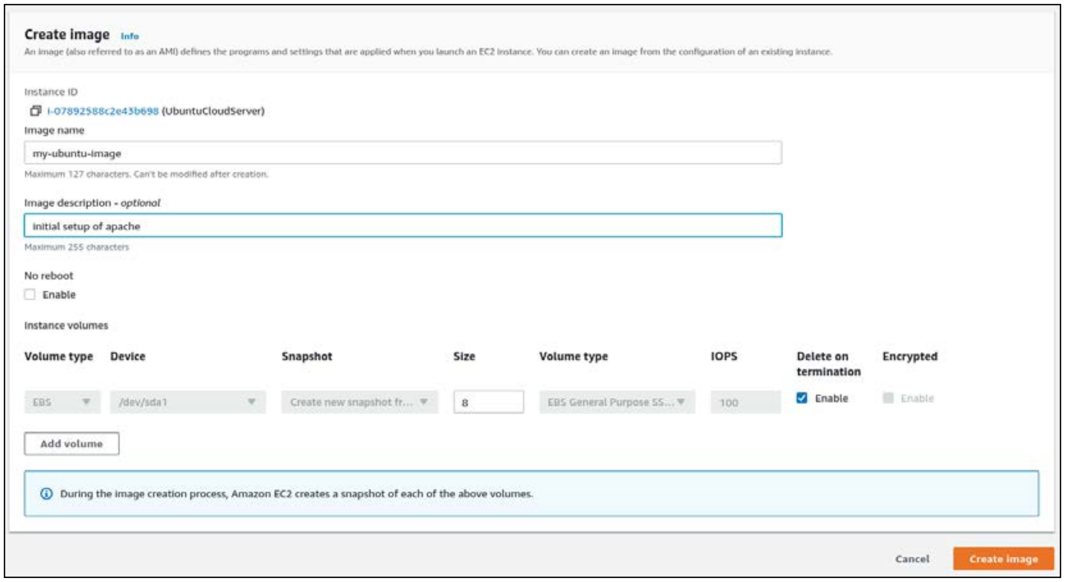


Figure 19-35: The Apache default web page running on an EC2 instance

Next, we can enter some details about our AMI. Give it a name and a description. This information will help others you work with understand what the image is for, and it can also help you remember why you've created the image later on down the road. Once you're finished, click **Create image** to continue:



**Create image** Info

An image (also referred to as an AMI) defines the programs and settings that are applied when you launch an EC2 instance. You can create an image from the configuration of an existing instance.

Instance ID

i-07892588c2e43b698 (UbuntuCloudServer)

Image name

my-ubuntu-image

Maximum 127 characters. Can't be modified after creation.

Image description - *optional*

initial setup of apache

Maximum 255 characters

No reboot

☐ Enable

Instance volumes

| Volume type | Device | Snapshot | Size | Volume type | IOPS | Delete on termination | Encrypted |
|---|---|---|---|---|---|---|---|
| EBS | /dev/sda1 | Create new snapshot fr... | 8 | EBS General Purpose SS... | 100 | ✓ Enable | Enable |

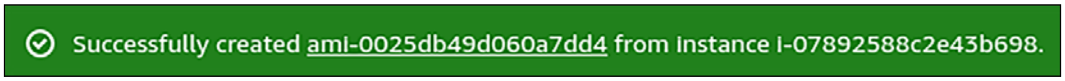
Add volume

During the image creation process, Amazon EC2 creates a snapshot of each of the above volumes.

Cancel Create image

Figure 19.36: Creating a new AMI

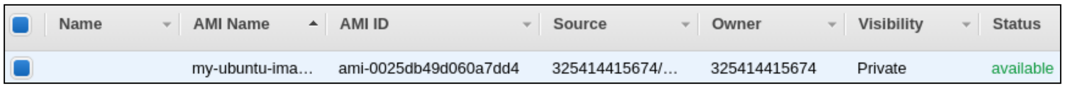
Believe it or not, that's all there is to it. The process of creating an AMI is very straightforward, with just a few steps. You should now see a confirmation screen, letting you know the image is in the process of being created:



Successfully created ami-0025db49d060a7dd4 from instance i-07892588c2e43b698.

Figure 19.37: Confirmation while creating a new AMI

If you click on the underlined text that contains the AMI ID, you'll be directed to the AMI section of the EC2 console, where it will show your image on the list:



| Name | AMI Name | AMI ID | Source | Owner | Visibility | Status |
|---|---|---|---|---|---|---|
| | my-ubuntu-ima... | ami-0025db49d060a7dd4 | 325414415674/... | 325414415674 | Private | available |

Figure 19.38: Our newly created AMI, available for use

In the AMI section of the EC2 console, you should see the list narrowed down to just the AMI we created just now. We only have one AMI anyway, unless you created multiple AMIs for practice. At the end, the **Status** column should read **available** if the AMI is ready for use. If not, give it some time, and refresh the page later. Sometimes it can take a few minutes. But with regard to creating an AMI, that's it!

Now that we have an AMI, how do we go about using it? Well, that's even easier actually. Simply right-click on the AMI on the list and click **Launch**. You'll see the same launch settings we worked through earlier when we originally created the instance, but this time, we're using our own AMI instead of the one provided to us. And now, we have our own custom AMI of Ubuntu with Apache built in that we can use to simplify our process a bit. Keep in mind, though, that our original instance is still stopped. You can return to your list of EC2 instances and start it by right-clicking on it, then clicking **Start**, but you don't have to, we're going to work through a fun automation example shortly.

In the next section, we're going to take a look at the concept of Auto Scaling.

## Automatically scaling Ubuntu EC2 deployments with Auto Scaling

If we maintain one or more servers for our organization, it's hard to predict sometimes what the demand will be on that server. In the case of a popular news site, some articles may be more popular than others, and if something goes viral online, then requests to our site can increase by orders of magnitude. In the past, keeping up with customer demand was a very tedious process, one that may result in having to purchase an entire new server with more powerful hardware. With our instance being in the cloud, we have more flexibility and can automate the process of bringing more servers online. And that's exactly what we're going to work on in this section.

Before we get started, keep in mind that we don't actually have a popular server in AWS; we only have a simple test server that's currently running Apache. We can simulate things to a point, but **Auto Scaling** is one of those things that requires a bit of practice to fully utilize. We will definitely get a working example here, though.

But another important thing to keep in mind is that the more instances we run, the higher the potential cost. We'll explore how to keep costs down in the next section, but as a general rule of thumb, delete whatever you're not using. As you've gone through examples in this chapter, we've set up our own EC2 instance. This is great: we were able to practice some concepts around AWS and put that to use. But if we leave something running that we don't need, we can have a surprise bill. It's a good idea to write yourself a reminder to delete everything in your test AWS account when we're done with the chapter, so you won't have to worry about that.

Continuing, one of the requirements of Auto Scaling is that we have an AMI ready that it will use to bring additional servers online. Since we've worked through creating an AMI in the previous section, we already have that requirement met. If you haven't already worked through the previous section, make sure you do so before we continue. The process of setting up Auto Scaling involves a handful of steps, and we'll work through each in their own sub-section within this chapter.

## Creating a launch template

Earlier in the chapter, we walked through the process of creating a new EC2 instance. We chose the option to launch an instance, and then configured various settings within multiple screens we worked through. We chose Ubuntu as our platform, added user data, and set an IAM role (among other things). What a **launch template** does is allow us to automate these choices. A launch template gives us the ability to automate the entire launch process.

On the left-hand menu of the EC2 console, there will be a link titled **Launch Templates**. Click on it. Once you do, you can click on the orange button labeled **Create launch template**. You'll then see a form you'll need to fill out, where you select all the defaults for the launch template. There's no screenshot on this page, because it's quite long and won't fit on one page. Instead, I'll include the relevant options below, with a short description and a recommendation regarding what to set the option to:

- **Launch template name**: This is simply a name for your launch template; set it to the name you feel is most appropriate. Note that this cannot contain spaces.
- **Template version description**: For the description, you can add some details that you think are relevant to the purpose of the launch template.
- **Amazon Machine Image** (**AMI**): Choose the AMI that you created in the previous section. It might be hard to find on the list, so you can copy the AMI ID (located in the AMI section of the EC2 console) and paste it here.

- **Key pair** (**login**): When you created the EC2 instance earlier, it had you create an OpenSSH key pair. If you drop down this list, that same key pair should be available. Choose that same key.
- **Instance type**: If you recall, we chose t2.micro as the instance type earlier. That's a good selection for this field as well, since t2.micro is eligible for the free tier.
- **Security groups**: Earlier, when we added a security group, we set it up to allow OpenSSH and Apache. Feel free to choose that same security group for this.

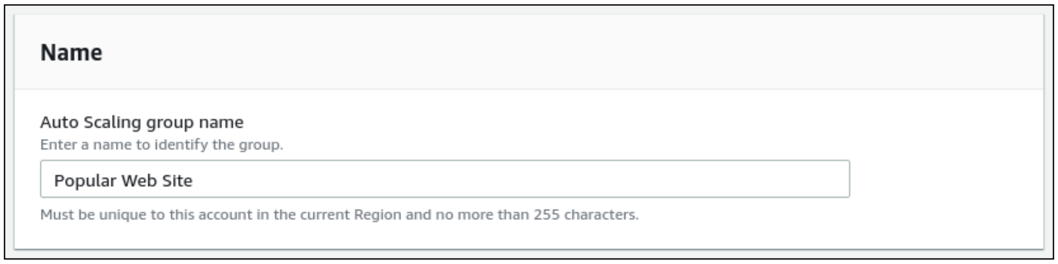
Near the bottom will be an additional section you can expand to give you the ability to configure advanced details. Underneath that, you'll have an option to set the IAM role for the template. For this, choose the same IAM role we created earlier to facilitate access via Session Manager.

With all of those details set, click **Create Launch Template** near the bottom of this screen. Now we have our launch template created and configured, and we can use it as part of the Auto Scaling feature we're in the process of building.

## Creating an Auto Scaling group

Our next requirement is to create an **Auto Scaling group**, which will be a shorter process than setting up the launch template. An Auto Scaling group is a logical group of instances that are related to the overall application. We will add our launch template to this group and use it to customize requirements such as how many instances to have online.

Back in the EC2 dashboard, you'll find an option for creating Auto Scaling groups in the menu on the left side of the screen, closer to the bottom. Once there, give it a name:



Name

Auto Scaling group name

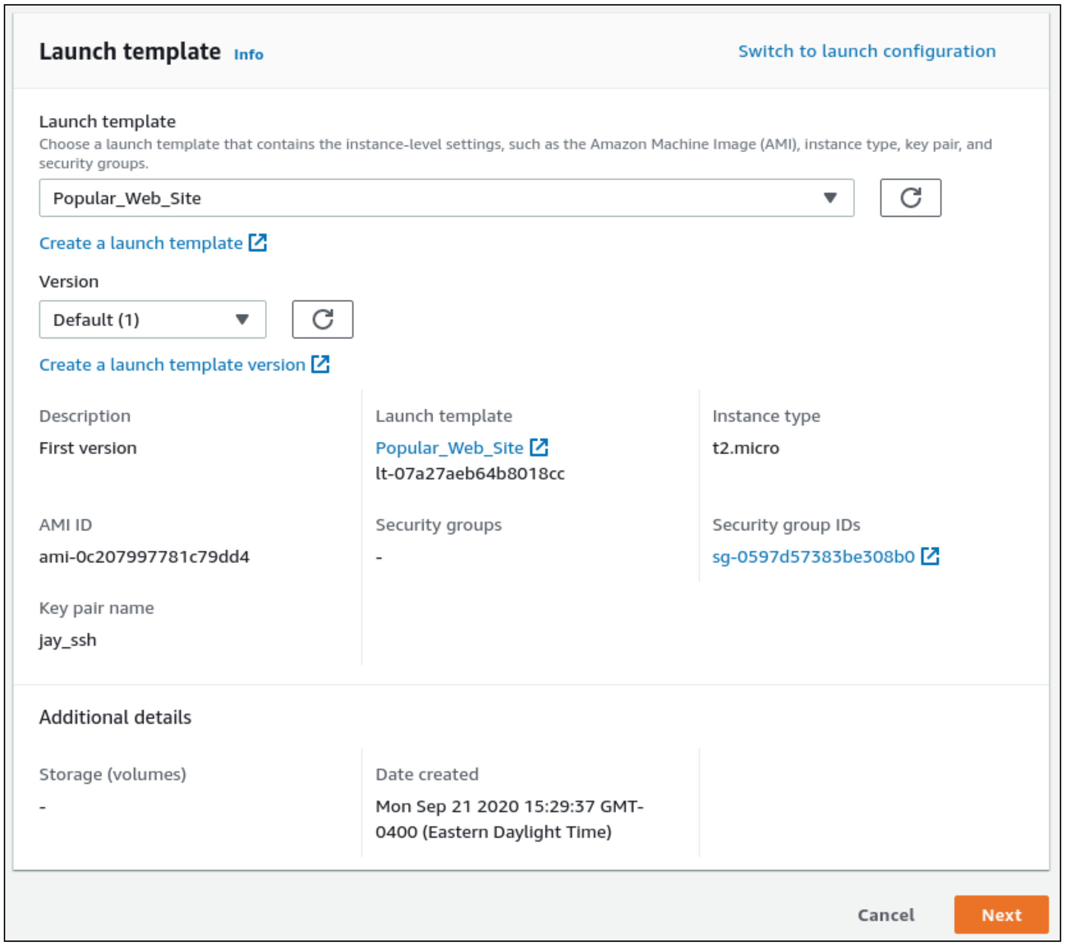
Enter a name to identify the group.

Popular Web Site

Must be unique to this account in the current Region and no more than 255 characters.

Figure 19.39: Naming the Auto Scaling group

Further down on that same screen, we will choose the launch template. Go ahead and do so, and then click **Next**:



Launch template Info

Switch to launch configuration

Launch template

Choose a launch template that contains the instance-level settings, such as the Amazon Machine Image (AMI), instance type, key pair, and security groups.

Popular_Web_Site

Create a launch template

Version

Default (1)

Create a launch template version

| | | |
|---|---|---|
| Description | Launch template | Instance type |
| First version | Popular_Web_Site lt-07a27aeb64b8018cc | t2.micro |
| AMI ID | Security groups | Security group IDs |
| ami-0c207997781c79dd4 | - | sg-0597d57383be308b0 |
| Key pair name | | |
| jay_ssh | | |

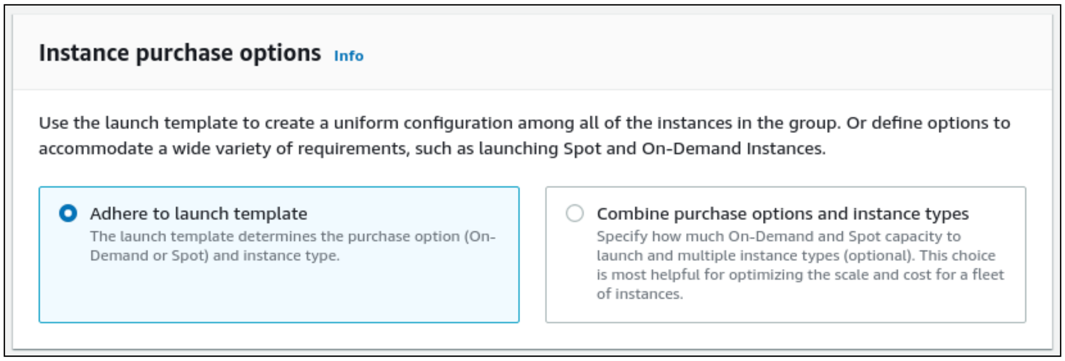
Additional details

| | |
|---|---|
| Storage (volumes) | Date created |
| - | Mon Sep 21 2020 15:29:37 GMT-0400 (Eastern Daylight Time) |

Cancel Next

Figure 19.40: Selecting a launch template for our new Auto Scaling group

On the next screen, you can leave the purchase option on its default, which is **Adhere to launch template**:



**Instance purchase options** Info

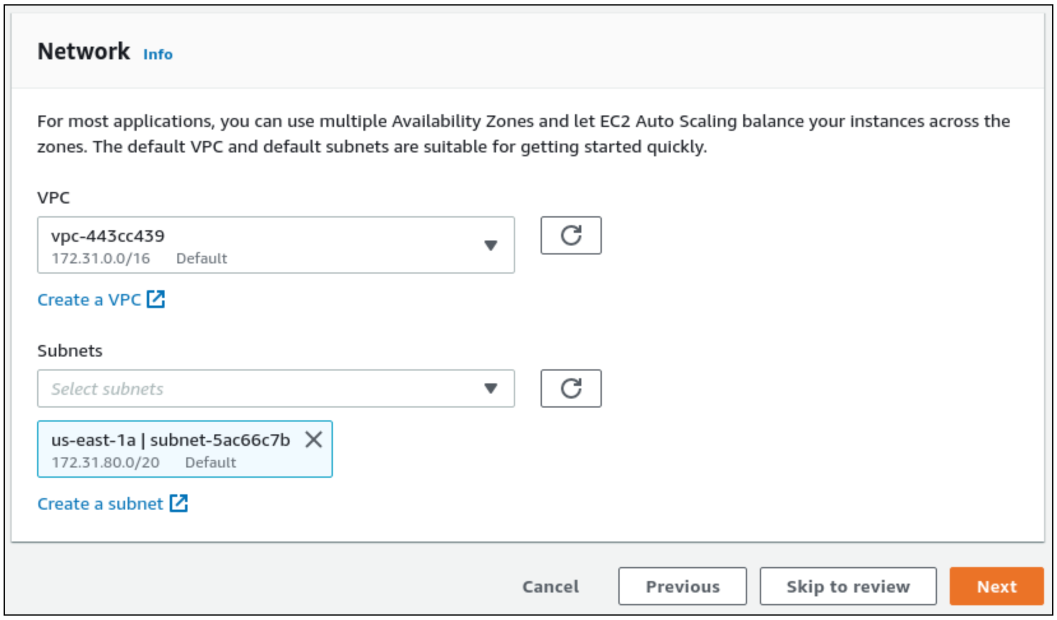
Use the launch template to create a uniform configuration among all of the instances in the group. Or define options to accommodate a wide variety of requirements, such as launching Spot and On-Demand Instances.

[✓ Adhere to launch template] The launch template determines the purchase option (On-Demand or Spot) and instance type.

Combine purchase options and instance types — Specify how much On-Demand and Spot capacity to launch and multiple instance types (optional). This choice is most helpful for optimizing the scale and cost for a fleet of instances.

Figure 19.41: Creating an Auto Scaling group (continued)

Underneath the purchase option on the same page, you can leave the VPC selection on its default, and then add one of the subnets on the list of those that are available. We won't be creating any resources available across availability zones in this walk-through, so it doesn't matter which you choose. Click **Next** to continue:



**Network** Info

For most applications, you can use multiple Availability Zones and let EC2 Auto Scaling balance your instances across the zones. The default VPC and default subnets are suitable for getting started quickly.

VPC: vpc-443cc439 172.31.0.0/16 Default

Create a VPC

Subnets: Select subnets

us-east-1a | subnet-5ac66c7b 172.31.80.0/20 Default

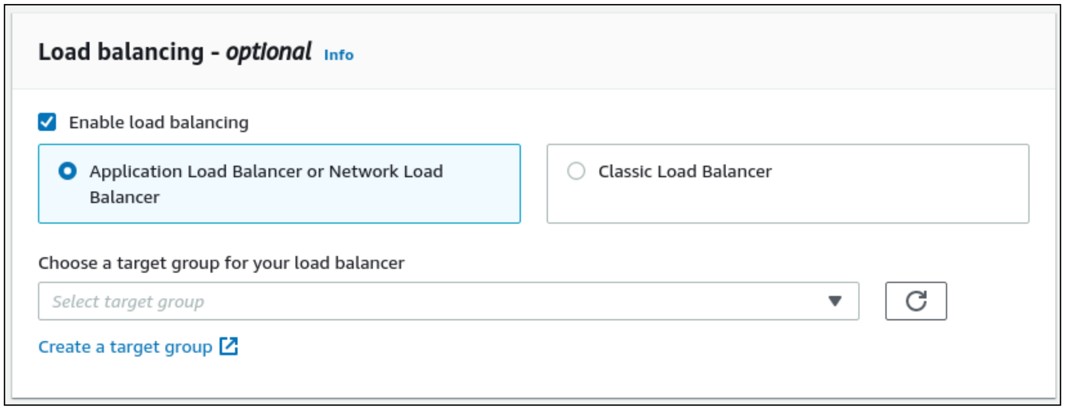
Create a subnet

Figure 19.42: Creating an Auto Scaling group (continued)

On the next page, we can choose to enable load balancing. A **load balancer** allows us to route clients between multiple servers, so the end-users only see one endpoint, but behind the load balancer we can have more than one server available to serve client connections. We'll want to utilize this, so be sure to check the box titled **Enable load balancing**.

There's also an option listed to create a **Classic Load Balancer**. A classic load balancer is only used in existing environments that were created in classic EC2 networks, but since this is a new account, we don't have existing environments to take into consideration.

The next option directly below that allows us to select a target group. However, we don't actually have one yet since we've never created one. You can click the **Create a target** group link on this page to create one now:



Load balancing - *optional* Info

[✓] Enable load balancing

● Application Load Balancer or Network Load Balancer
○ Classic Load Balancer

Choose a target group for your load balancer

Select target group

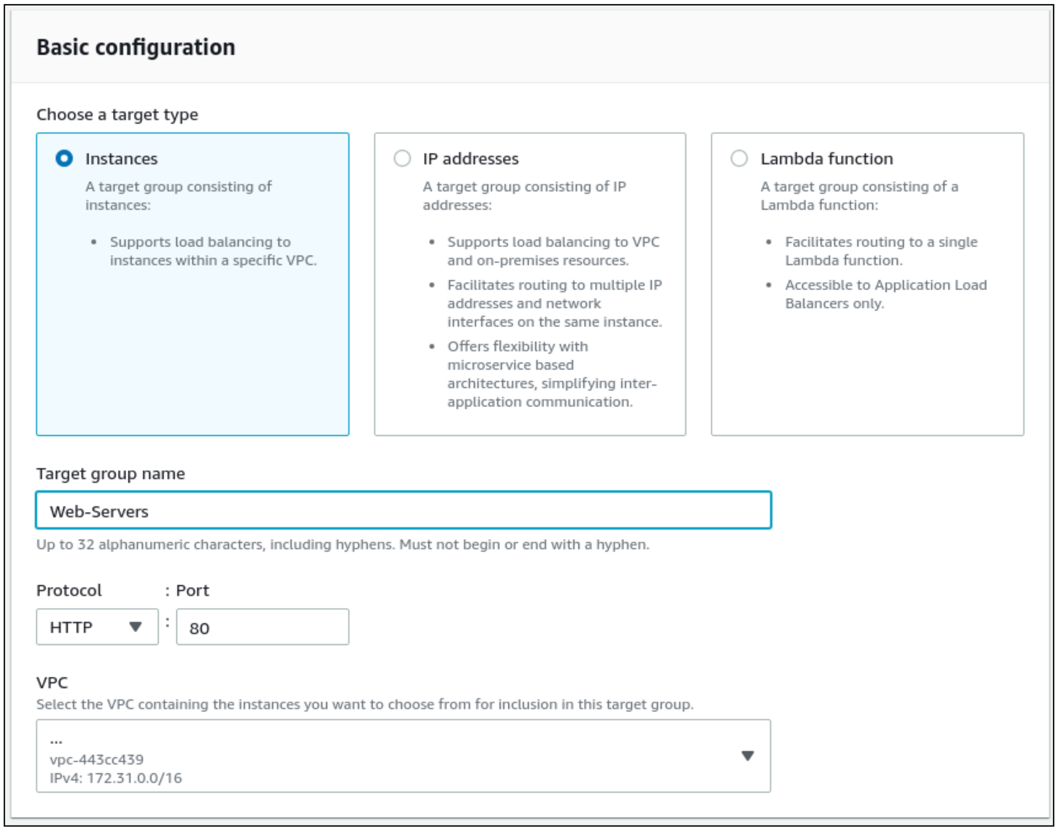
Create a target group

Figure 19.43: Creating an Auto Scaling group (continued)

This will cause a new window to appear, taking us directly to the place within the AWS console that allows us to manage our target groups. A **Target Group** is a group of instances that are responsible for serving our application. In our case, we installed Apache in our EC2 instance earlier, and we used that instance to create an AMI. Following along with this example, a target group for us could be a group of web servers created via this AMI, each of which can serve our web page to clients. We didn't actually alter the default Apache web page, but the example still works.

For the first screen while creating our target group, we should ensure that the **Instances** box is selected, then we can give the target group a name.

That's all we need to do here, so go ahead and click **Next** to continue:


Basic configuration

Choose a target type

[✓ Instances]
A target group consisting of instances:
- Supports load balancing to instances within a specific VPC.

IP addresses
A target group consisting of IP addresses:
- Supports load balancing to VPC and on-premises resources.
- Facilitates routing to multiple IP addresses and network interfaces on the same instance.
- Offers flexibility with microservice based architectures, simplifying inter-application communication.

Lambda function
A target group consisting of a Lambda function:
- Facilitates routing to a single Lambda function.
- Accessible to Application Load Balancers only.

Target group name: Web-Servers
Up to 32 alphanumeric characters, including hyphens. Must not begin or end with a hyphen.

Protocol : Port
HTTP : 80

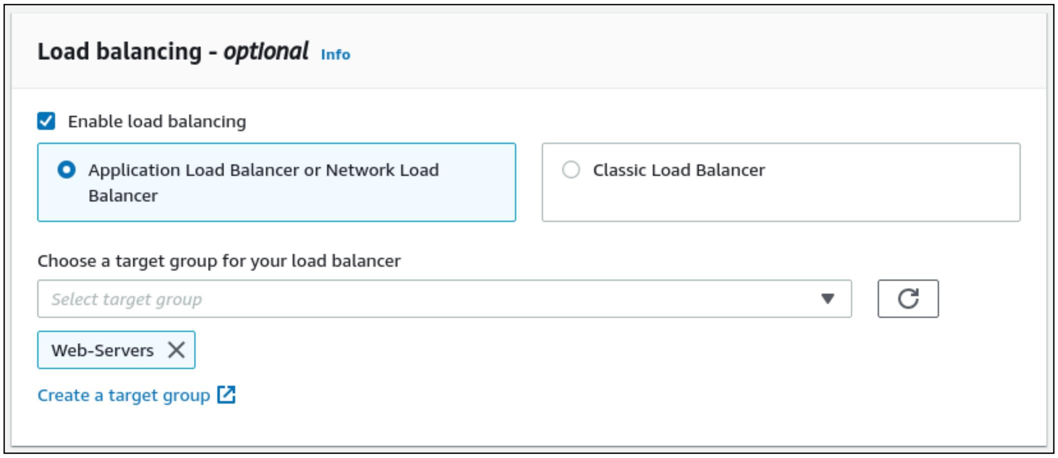
VPC
Select the VPC containing the instances you want to choose from for inclusion in this target group.
...
vpc-443cc439
IPv4: 172.31.0.0/16

Figure 19.44: Creating a target group

On this page, ensure **Instances** is selected and give the target group a name.

On the next screen, we will have an option to add existing EC2 instances to our target group. Let's leave everything blank, and not add any targets. We can click **Create target group** to finalize the process.

Back on the page we were on while creating the Auto Scaling group, we can now select the target group that we created in the previous step:



Load balancing - *optional* Info

[✓] Enable load balancing

● Application Load Balancer or Network Load Balancer

○ Classic Load Balancer

Choose a target group for your load balancer

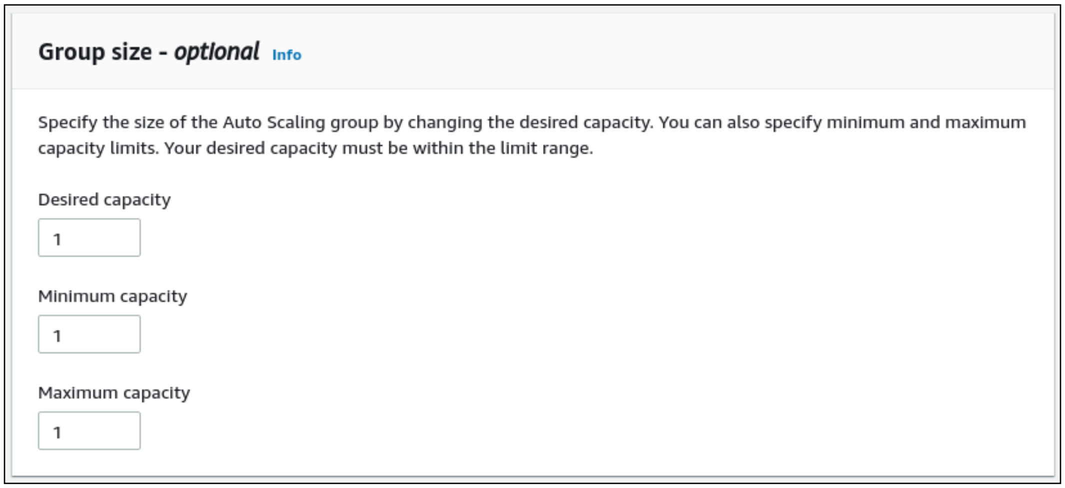
Select target group

Web-Servers

Create a target group

Figure 19.45: Creating an Auto Scaling group (continued)

Next, we can set the size and scaling policies. This is where the magic happens: we can choose the minimum number of instances to have running at any one time, the maximum number of instances we will allow the application to scale up to. We can leave each field at **1** for now:



Group size - *optional* Info

Specify the size of the Auto Scaling group by changing the desired capacity. You can also specify minimum and maximum capacity limits. Your desired capacity must be within the limit range.

Desired capacity: 1

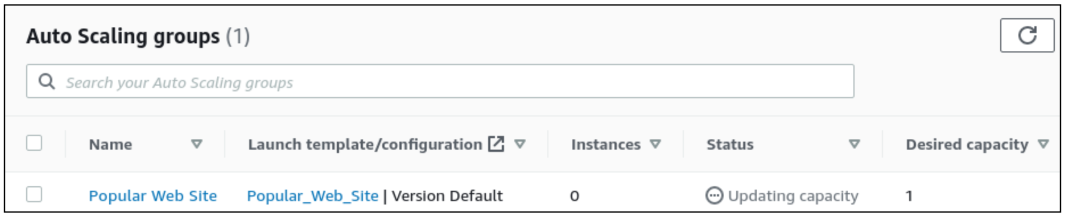
Minimum capacity: 1

Maximum capacity: 1

Figure 19.46: Creating an Auto Scaling group (continued)

For the remainder of the configuration on this page, we can ignore the rest and continue along. In fact, you can keep the defaults for each remaining screen, clicking **Next** each time, and then at the end click **Create Auto Scaling group**.

At this point, our Auto Scaling group is created, and we should have everything ready to go. At first, we'll have **0** instances within this group, so we'll see the current number of instances shown as **0**:



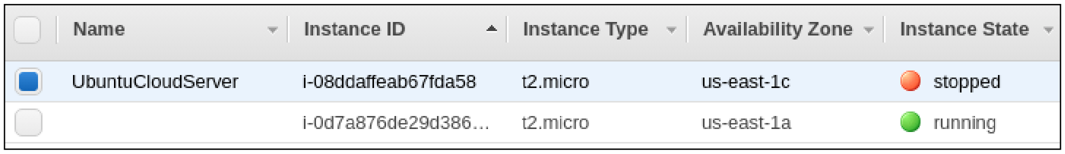
**Auto Scaling groups** (1)

Search your Auto Scaling groups

| Name | Launch template/configuration | Instances | Status | Desired capacity |
|---|---|---|---|---|
| Popular Web Site | Popular_Web_Site \| Version Default | 0 | Updating capacity | 1 |

Figure 19.47: Auto Scaling groups, showing our newly created configuration

After it finishes updating capacity, it will automatically spin up a new EC2 instance in order to meet our requirement of always having one instance online. If we check our list of EC2 instances, we should have a new one on the list now:



| Name | Instance ID | Instance Type | Availability Zone | Instance State |
|---|---|---|---|---|
| UbuntuCloudServer | i-08ddaffeab67fda58 | t2.micro | us-east-1c | 🔴 stopped |
| | i-0d7a876de29d386… | t2.micro | us-east-1a | 🟢 running |

Figure 19.48: A new instance was created as part of our Auto Scaling configuration

As you can see in the screenshot, the original EC2 instance has a state of **stopped**. We stopped it earlier so that we could create an AMI of that instance. In my case, I never started that instance again after creating the AMI, so it's completely stopped. The Auto Scaling configuration went ahead and created a new instance, because its requirement of having at least one instance running was not met.

Before we implement the final component, we need for everything to function properly, let's take a moment to understand the value that we already have in place. If we were to increase the number of desired instances within the Auto Scaling group, then it would immediately spin up a new instance for us. Although advanced usage is beyond the scope of this book, we can set this up to automatically happen when the CPU of our instance gets to a certain point, which can trigger Auto Scaling to bring another one online.

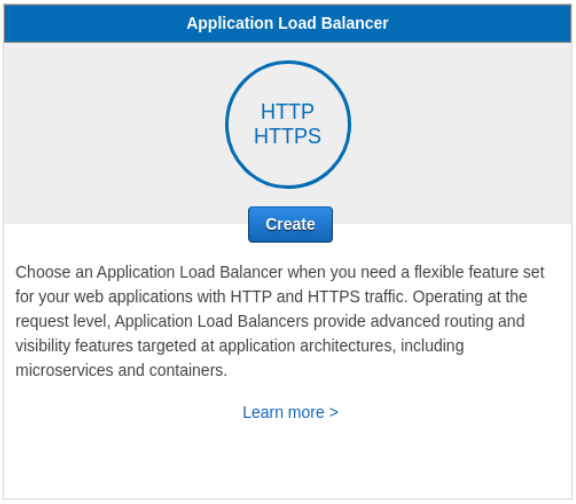
With just a single instance, we don't have Auto Scaling in play yet, but we can easily enable that. Another benefit that we get automatically is auto healing, and we have that benefit even with a single instance. If something were to happen to our only running instance, Auto Scaling will bring a new one online to replace it automatically.

In this situation, the website will be down for several minutes while the new instance is being created, but a bit of downtime is certainly better than having to manually replace the server ourselves. If we set the desired instances to a number higher than 1, a user will not notice anything if one of the servers goes down, the other will take care of the load while the new one comes up. These are amazing benefits to be able to take advantage of and will give us additional peace of mind right away. To test this yourself, feel free to delete the running EC2 instance by right-clicking it and then clicking **Terminate**. The instance should then get terminated, and a new one should appear within several minutes to replace it.

Next, we can implement the final piece of our puzzle by creating a load balancer.

## Creating a load balancer

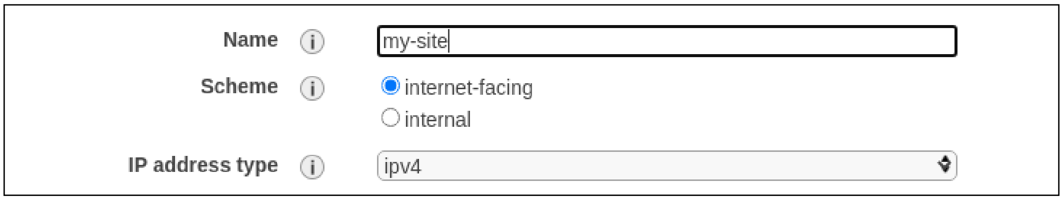
A load balancer, as mentioned earlier, routes our clients to a server in order to facilitate their requests. Having a load balancer is what allows us to be able to withstand one or more EC2 instances going down and becoming inaccessible, the load balancer will route users to an instance that is working fine, while Auto Scaling takes over and replaces failed instances for us. To get started, you can click on the **Load Balancers** section of the EC2 console. Once there, click on the **Create** button located on the first of three squares that appears:



Choose an Application Load Balancer when you need a flexible feature set for your web applications with HTTP and HTTPS traffic. Operating at the request level, Application Load Balancers provide advanced routing and visibility features targeted at application architectures, including microservices and containers.

Figure 19.49: Creating a load balancer

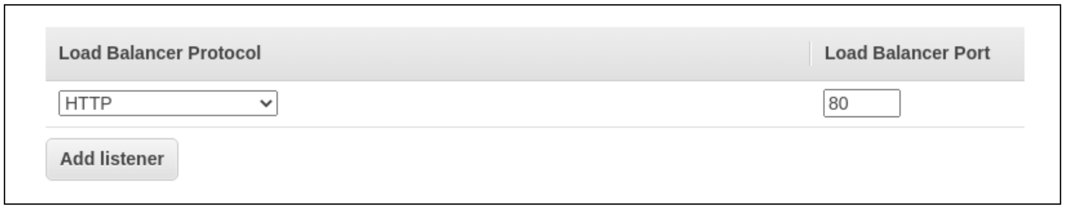
Next, give the load balancer a name. Note that you cannot have spaces here. The other two settings can remain at their defaults:



| | |
|---|---|
| Name | my-site |
| Scheme | [✓ internet-facing] internal |
| IP address type | ipv4 |

Figure 19.50: Creating a load balancer (continued)

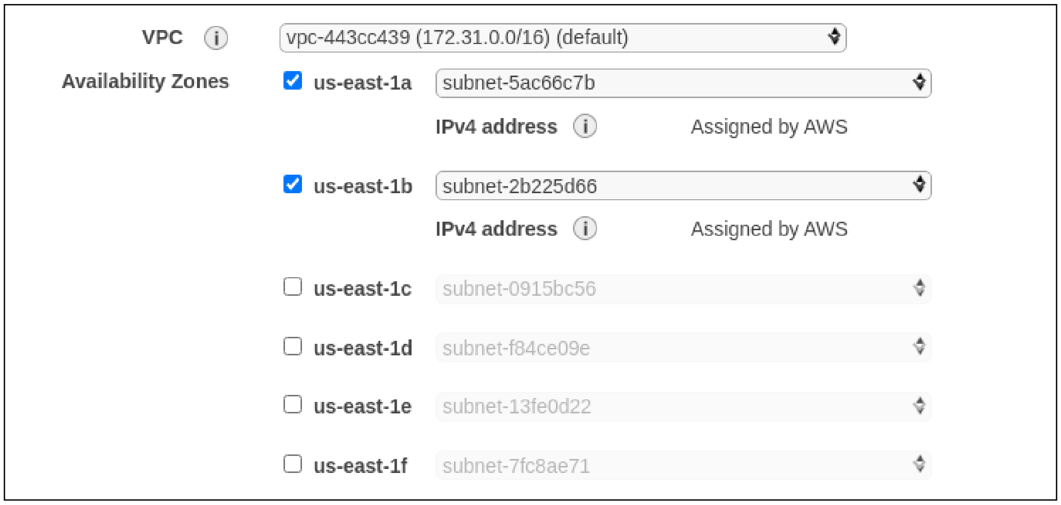
Beneath that section on the same page, we can set the protocol and port for the load balancer. Port `80` is fine for our tests, although we would want to consider implementing TLS and utilizing port `443` if this were a production application:



| Load Balancer Protocol | Load Balancer Port |
|---|---|
| HTTP | 80 |

Figure 19.51: Creating a load balancer—Load Balancer Protocol

Further down, we can choose **Availability Zones** for our application. You can just leave the defaults and choose the first two subnets. Click **Next: Configure Security Settings** to continue:



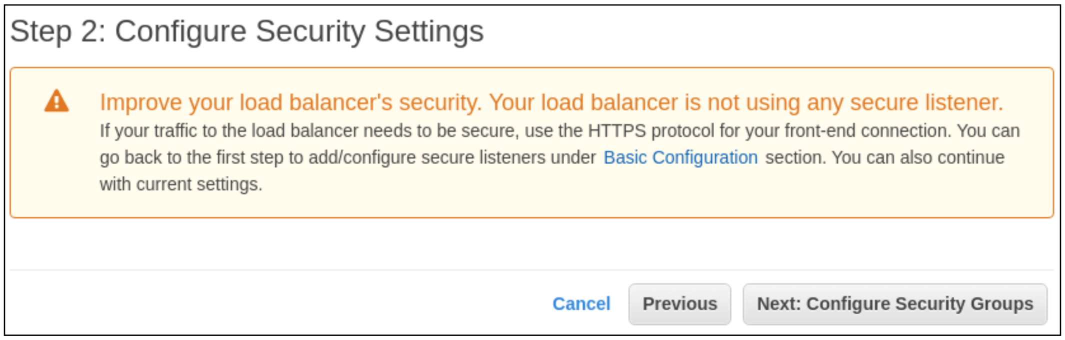
VPC: vpc-443cc439 (172.31.0.0/16) (default)

Availability Zones:

| | | |
|---|---|---|
| ✓ us-east-1a | subnet-5ac66c7b | IPv4 address: Assigned by AWS |
| ✓ us-east-1b | subnet-2b225d66 | IPv4 address: Assigned by AWS |
| us-east-1c | subnet-0915bc56 | |
| us-east-1d | subnet-f84ce09e | |
| us-east-1e | subnet-13fe0d22 | |
| us-east-1f | subnet-7fc8ae71 | |

Figure 19-52: Creating a load balancer—Availability Zones

The next page will warn you about not using HTTPS for our application, which we can safely ignore since this is just a test. Click **Next: Configure Security Groups** to move on:

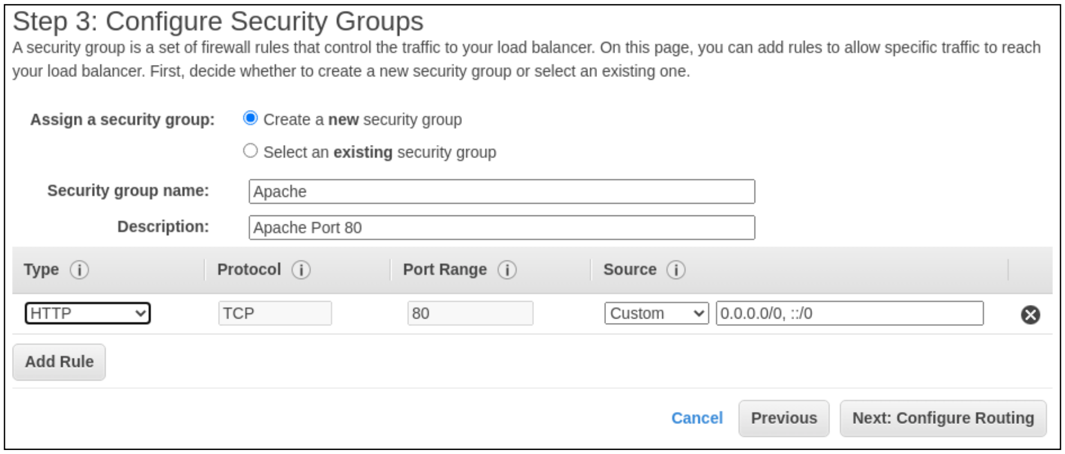


Step 2: Configure Security Settings

Improve your load balancer's security. Your load balancer is not using any secure listener.
If your traffic to the load balancer needs to be secure, use the HTTPS protocol for your front-end connection. You can go back to the first step to add/configure secure listeners under Basic Configuration section. You can also continue with current settings.

Figure 19.53: Creating a load balancer—Configure Security Settings

Continuing on to the next page, we're able to choose one or more security groups to provide access to the application. We're able to use the existing security group here that we created earlier, but since we enabled OpenSSH on that security group, I don't recommend we use it as we don't want to enable OpenSSH on everything. Instead, we can create a new security group (as shown in the following screenshot) to allow access to port 80 for Apache. After adding those values, click **Next: Configure Routing** to proceed:



Step 3: Configure Security Groups

A security group is a set of firewall rules that control the traffic to your load balancer. On this page, you can add rules to allow specific traffic to reach your load balancer. First, decide whether to create a new security group or select an existing one.

Assign a security group: [✓ Create a **new** security group] Select an **existing** security group

Security group name: Apache

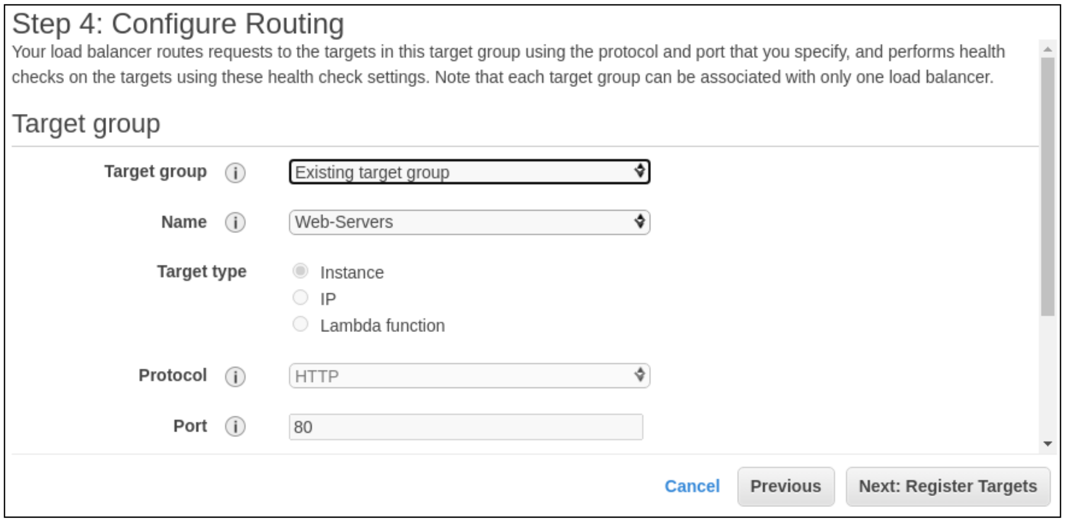
Description: Apache Port 80

| Type | Protocol | Port Range | Source |
|---|---|---|---|
| HTTP | TCP | 80 | Custom 0.0.0.0/0, ::/0 |

Figure 19.54: Creating a load balancer—Configure Security Groups

For the fourth step in the process of creating the load balancer, we'll select our target group and set that to point to the target group we've already set up. This means that if a user goes to access our site, the load balancer will forward them to the target group, which will in turn send them to one of its instances.

We can leave the rest as it is and click **Next: Register Targets** to go to the next step, and accept the defaults and click through the next sections that appear after that:



Step 4: Configure Routing

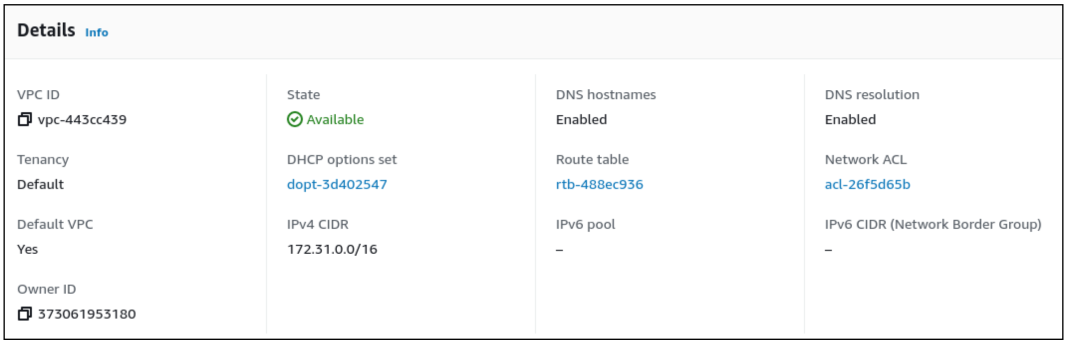
Your load balancer routes requests to the targets in this target group using the protocol and port that you specify, and performs health checks on the targets using these health check settings. Note that each target group can be associated with only one load balancer.

Target group

| | |
|---|---|
| Target group | Existing target group |
| Name | Web-Servers |
| Target type | [✓ Instance] IP Lambda function |
| Protocol | HTTP |
| Port | 80 |

Figure 19.55: Creating a load balancer – Configure Routing

At this point, the load balancer is created, and we have almost everything we need. The only thing left now is to ensure that our instances within our Auto Scaling configuration can be reached via port 80, and to do that, we'll need to add the IP range of our VPC into our security group. To find this value, we can access the VPC service within the list of available AWS services, and find our default VPC. After you access the VPC console, you should see an option titled **Your VPCs** in the menu on the left. After you click on that, you can then click on the VPC ID that's on the list in the middle (there should only be one), which will show you the details for the VPC:



Details Info

| | | | |
|---|---|---|---|
| VPC ID<br>vpc-443cc439 | State<br>Available | DNS hostnames<br>Enabled | DNS resolution<br>Enabled |
| Tenancy<br>Default | DHCP options set<br>dopt-3d402547 | Route table<br>rtb-488ec936 | Network ACL<br>acl-26f5d65b |
| Default VPC<br>Yes | IPv4 CIDR<br>172.31.0.0/16 | IPv6 pool<br>– | IPv6 CIDR (Network Border Group)<br>– |
| Owner ID<br>373061953180 | | | |

Figure 19.56: Creating a load balancer – VPC details

Notice the **IPv4 CIDR** that is listed within the details. Copy that entire value, as we'll need it shortly. Next, access the EC2 service, and access the **Security groups** section, where we should find the first security group we added earlier in the chapter. Click on that **security group**, and then click the button labeled **Edit Inbound Rules** and you should be able to add the VPC IP range to be accessible via port `80`. The new line should look similar to the following:

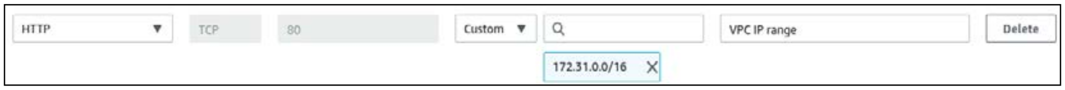
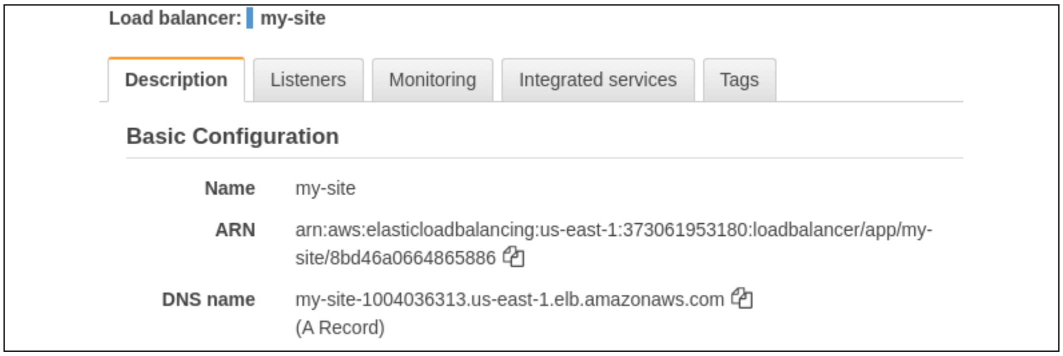


Figure 19.57: Adding a new security group entry to allow VPC traffic

With that final setting implemented, our application should be accessible to our users via the load balancer. To test that everything is working correctly, we can access the **Load balancer** section of the EC2 console, which is available in the menu at the left, to see details regarding the load balancer that is currently in front of our target group:



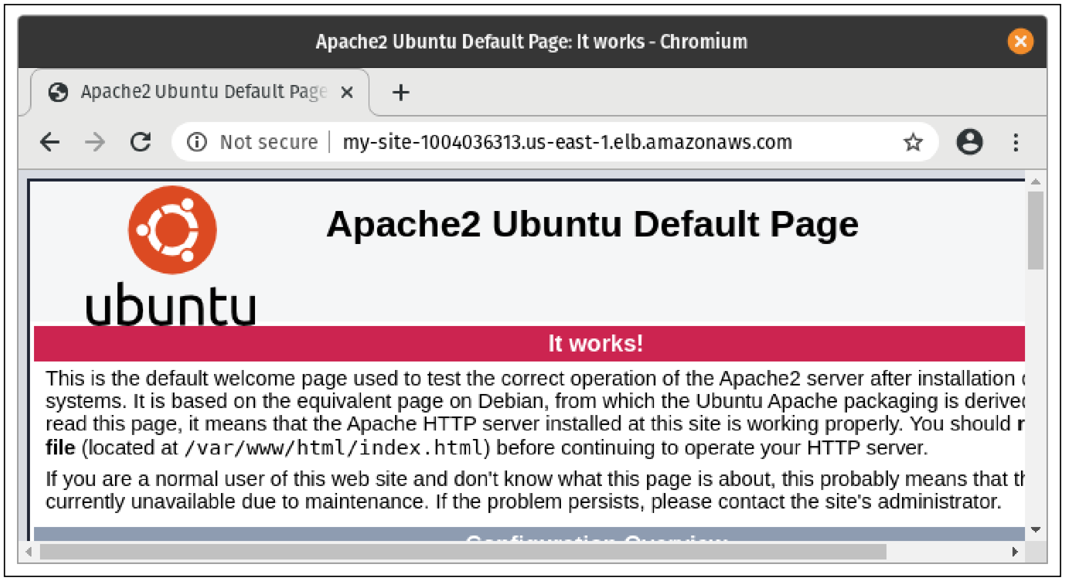
Load balancer: my-site

Basic Configuration

| | |
|---|---|
| Name | my-site |
| ARN | arn:aws:elasticloadbalancing:us-east-1:373061953180:loadbalancer/app/my-site/8bd46a0664865886 |
| DNS name | my-site-1004036313.us-east-1.elb.amazonaws.com (A Record) |

Figure 19.58: Viewing load balancer details

Note the **DNS name** that appears when you select the load balancer. If you copy and paste that into a browser, the default Apache page should appear:


Apache2 Ubuntu Default Page: It works - Chromium
Apache2 Ubuntu Default Page
Not secure
my-site-1004036313.us-east-1.elb.amazonaws.com
ubuntu

Apache2 Ubuntu Default Page

It works!

This is the default welcome page used to test the correct operation of the Apache2 server after installation
systems. It is based on the equivalent page on Debian, from which the Ubuntu Apache packaging is derive
read this page, it means that the Apache HTTP server installed at this site is working properly. You should r
**file** (located at `/var/www/html/index.html`) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that t
currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

Figure 19.59: Viewing load balancer details

Congratulations! At this point, you've created a complete load-balanced solution in AWS that will automatically heal from failures. Feel free to experiment a bit more, and then when you're finished, consider removing the test components we've created in this chapter to avoid a cost later on.

Speaking of cost, in the next section, we'll talk a bit further about how to manage costs and keep our bills under control.

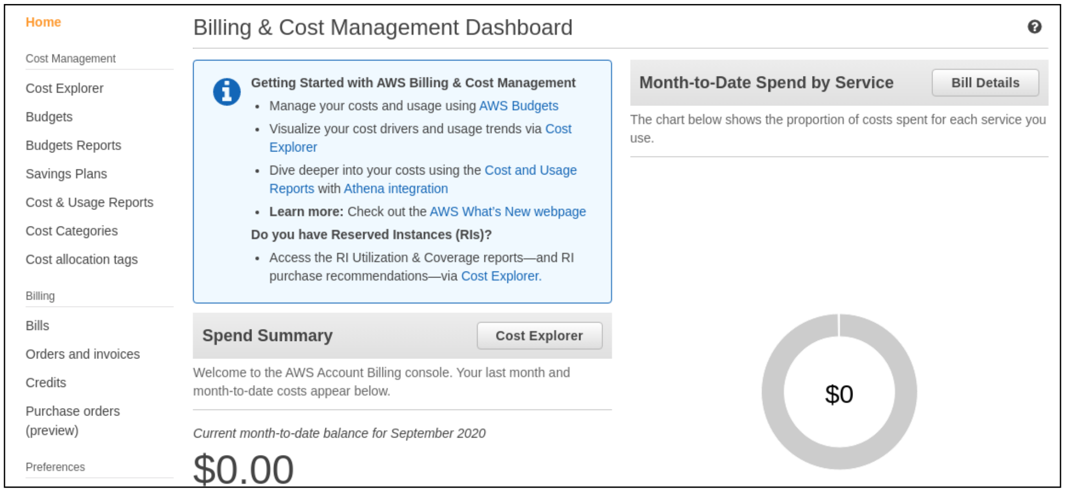
## Keeping costs down: understanding how to save money and make cost-effective decisions

As you just saw, there were many components and configurations we had to implement in order to implement a load-balanced solution in AWS. As we grow our AWS infrastructure and implement more solutions, we should also keep an eye on our bill. Although we can utilize the free tier for now, production applications will likely need more powerful instances than what the free tier will provide, and the free tier itself won't last forever. Not only that, but we should also know how to check how our bill is trending to make sure we don't accidentally implement something that is expensive or waste money by running something we no longer need.

In this section, we'll explore some concepts around billing. Although it's beyond the scope of this chapter to do a complete deep-dive into the world of billing, the subsections that follow will provide you with essential advice needed to help prevent unexpected charges.

## Viewing billing information

Within AWS, the billing section is its own service along with others, such as EC2 and S3. You can find the **Billing** area from the service list. Once there, you'll be shown the **Billing & Cost Management Dashboard**, which will show you your current expenses and allow you to see current and past billing information:



Getting Started with AWS Billing & Cost Management

- Manage your costs and usage using AWS Budgets
- Visualize your cost drivers and usage trends via Cost Explorer
- Dive deeper into your costs using the Cost and Usage Reports with Athena integration
- Learn more: Check out the AWS What's New webpage

Do you have Reserved Instances (RIs)?

- Access the RI Utilization & Coverage reports—and RI purchase recommendations—via Cost Explorer.

The chart below shows the proportion of costs spent for each service you use.

Welcome to the AWS Account Billing console. Your last month and month-to-date costs appear below.

Current month-to-date balance for September 2020

$0.00

Figure 19.60: Viewing the Billing & Cost Management Dashboard

Since the account I'm working with currently was just created a week or so ago, and I'm only using instances in the free tier, I have no costs currently. But if I did incur charges, I'd see the current balance on the main page of the dashboard. In addition, I'll receive the same information in a monthly statement that is sent to the primary email account.

However, I don't recommend waiting for the bill to arrive before checking your totals. To effectively manage billing, you should check this dashboard manually, which might enable you to catch an error before the end of the month, which might save you money. The links on the left will give you additional billing information, as well as providing a way of accessing previous bills.

## Adding a billing alert

In the previous section, I recommended that you check the bill regularly instead of only when the monthly invoice arrives. While that is good advice (if I do say so myself), the reality is that server administrators are busy and may not remember to check the bill on a regular basis. This is why we typically set up system alerts to notify us when our servers are encountering an issue. Similarly, we can actually set up an alert inside our AWS account that can notify us if our bill gets too high. In fact, even though the AWS account we've been working with was likely only created as a test account, it's especially important that we add a billing alert so we can be alerted if any of our experiments were misconfigured in such a way that we're incurring costs we didn't intend to.

In the *Further reading* section at the end of the chapter, I have included a link to AWS documentation that details how to set up billing alerts, and I definitely recommend you enable them. If you're using an AWS account for your organization to run actual production servers, it's a good idea to enable billing alerts there as well. In fact, be sure to create billing alerts in every AWS account you manage. Another important element to consider is removing backups that aren't needed anymore, which can also lower costs.

## Removing unneeded backups

This may seem like a no-brainer, but you'd be surprised—every organization I've ever worked for or consulted with in regards to AWS has run into an issue where backups get out of control and generate large costs. My personal record for witnessing a waste of money in this manner is one company that had over 23,000 unnecessary snapshots in their account, sitting around for over 5 years. I don't remember the exact dollar amount, but this error cost them thousands of dollars a month for years.

Backups themselves are extremely important though, and the previous example might've been understandable if the organization had legal requirements that forced them to retain all backups for a specific period (such as 5 years). And backups, as you well know by now, are essential if the organization runs into some sort of issue and needs to restore something from the past. But the general rule of thumb here is to ensure that when the day comes to add an automatic backup feature, you also implement (and regularly test) an automatic cleanup procedure as well.

## Running EC2 instances only when they're needed

Many organizations do business around the clock and all year round, while others conduct business only during daytime business hours. While it might not be immediately apparent why this matters, consider the fact that you don't get charged for an EC2 instance while it's powered off. You do get charged for things like storage regardless of its state, but you aren't charged for the instance itself if it is not running. If your organization has a server that is only used during certain hours, consider stopping the instance outside of that time period and then starting it back up when it's needed.

There's additional functionality in AWS that allows you to automatically schedule an instance to run only during certain hours, so there are ways to ensure your infrastructure is available when it's needed. Make sure you keep this in mind as you navigate AWS, if you're creating a new server, is it going to be necessary to make it run 24/7, or is it only necessary to run it at certain times? You'd be surprised how much money this may save you.

## Stopping or terminating unneeded EC2 instances

Similar to the advice in the previous section about having instances run only when necessary, consider deleting instances completely if they're not needed at all. Unless you're utilizing a very specific type of instance, you won't be billed for an EC2 instance that doesn't exist. Make a habit of deleting things when you're done with them. If you think you may need a server again in the future but you're not sure, consider creating an AMI of the server, and then removing it. Although the AMI itself will have a cost, it's going to be less than running an actual server.

This advice isn't exclusive to EC2 instances, other services within AWS will cause you to incur costs if you don't clean them up. Make sure to keep an eye on other components and services that charge by usage, such as RDS, S3, EBS volumes, and so on.

So, there you have it—with some basic advice, you should be able to keep your billing under control. And even if you make a mistake, as long as you've configured billing alerts, you should be notified and be able to correct the problem quickly. I can understand if some of the billing advice was overwhelming, but you'll get used to it. AWS has a lot of components that can make up a bill, but as long as you set up alerts and delete items you're not using, you shouldn't have any problems.

We've gone over quite a bit in this chapter, but where should you go from here? In the next section, I'll provide some advice for continuing your learning and taking your AWS skills to the next level.

## Taking cloud further: additional resources to grow your knowledge

AWS is a huge service, and we haven't even scratched the surface of the platform in this chapter. We've just created a simple load-balanced application in an earlier section, and we'll even learn how to automate creating cloud resources in the next chapter. But if you've found this chapter fun and want to work with AWS more and enhance your skills, I thought I'd provide some additional advice that will hopefully help you do that.

### Online training and labs

There is quite a bit in the way of online resources to expand your knowledge. Some of these resources are free, such as a section of the AWS web site that provides free hands-on training: `https://aws.amazon.com/training/intro-to-aws-labs-sm/`.

While you may already be aware of the value of YouTube when it comes to training videos, it's a great source of knowledge. (And you may have even stumbled across my YouTube page, over at LearnLinux.tv). There are many videos on YouTube that can provide training, but that's not the only source of video content; Packt Publishing features video training courses as well, in addition to Udemy, which also has some great content.

Overall, there's no shortage of training materials available, but I'd recommend starting with the free training provided by AWS that I've mentioned above, as well as the additional training content that Amazon makes available at `https://aws.training`.

### Certification

While it will take a bit of work, achieving one or more certifications in AWS will lead you down a path where you'll learn the platform in much greater detail. In addition, individuals with AWS certifications are in high demand in the IT industry, so achieving certification is a good idea all around. I recommend looking into the **AWS Certified Cloud Practitioner** credential, which is a more entry-level certification that is approachable to those just starting out.

After you grow your expertise, you may want to consider the **AWS Certified Sysops Administrator** credential, which is more challenging but will boost your knowledge even further.

## Keep experimenting and learning

Even if you don't decide to achieve a certification, keep experimenting with the platform. Getting your hands on the technology and making use of it on a regular basis is usually the best way to learn and keep your skills sharp. Try to create additional types of infrastructure, build and rebuild test instances, and above all have fun. For many people, there's no greater way to learn something than to get your hands dirty and experience it. I also recommend you follow any blogs around cloud computing and related technologies as well.

## AWS documentation

The documentation provided by AWS is well written and very detailed. You can learn everything you need to know from the AWS documentation alone. The documentation pages are above and beyond the typical level of quality you would expect from such a large service; Amazon takes the AWS documentation seriously. The documentation pages are available here: `https://docs.aws.amazon.com/index.html`.

New resources for learning about AWS are being created on a regular basis, so keep your eyes open for new books, training videos, and more as you study the platform. I think you'll have a lot of fun taking your skills to the next level and exploring everything the AWS platform has to offer.

## Summary

This chapter has been one of the most involved in the entire book so far, and you've accomplished a lot. During the course of this chapter, you learned about AWS, set up your own cloud server, set up Auto Scaling to ensure that your server is able to automatically heal from disasters, and even set up a load balancer to enable routing between multiple instances. Make sure you take some time to let all this knowledge sink in before continuing on, and I also recommend you spend some additional time with AWS before moving on to the next chapter.

Speaking of the next chapter, we're going to work with AWS again, but this time, we're going to focus on learning Terraform, which is an awesome tool that will enable us to automate the building of our cloud resources from the ground up. It's going to be a lot of fun.

## Further reading

- Amazon Elastic Compute Cloud documentation: `https://docs.aws.amazon.com/ec2/index.html`
- Security groups for your VPC: `https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html`
- Auto Scaling documentation: `https://docs.aws.amazon.com/autoscaling/index.html`
- Amazon Machine Images (AMIs): `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html`
- Amazon Elastic Kubernetes Service documentation: `https://docs.aws.amazon.com/eks/index.html`
- AWS Billing Alert documentation: `https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html`

# 20
# Automating Cloud Deployments with Terraform

The previous chapter was especially fun: we were able to deploy Ubuntu in the cloud, utilizing **Amazon Web Services** (**AWS**). Deploying infrastructure in the cloud is very powerful and allows us to accomplish things that are not normally possible (or are very tedious) with physical infrastructure. We can spin-up Ubuntu instances in minutes, and even set up auto healing to cover us in situations that would normally result in complete service disruption.

This time around, we're going to work with cloud deployments again, and check out an awesome tool called **Terraform** that will allow us to automate the provisioning of our cloud resources. We've already explored the concept of automation back in *Chapter 15*, *Automating Server Configuration with Ansible*, when learned the basics of Ansible. Terraform allows us to take our automation to the next level and even interact with providers such as AWS directly.

In this chapter, we'll explore the following concepts:

- Why it's important to automate your infrastructure
- Introduction to Terraform and how it can fit within your workflow
- Installing Terraform
- Automating an EC2 instance deployment
- Managing security groups with Terraform
- Using Terraform to destroy unused resources
- Combining Ansible with Terraform for a full deployment solution

Why automate the building of our infrastructure? There are many benefits of doing so, and we'll take a look at some of those benefits in the next section.

## Why it's important to automate your infrastructure

Automation with regards to infrastructure is an expansive topic, and it easily deserves a book of its own. In fact, there are not only books dedicated to it but entire online courses as well. There are many different utilities you can use, each with their own pros and cons. We have configuration management tools, such as Ansible, Chef, and Puppet. We looked at Ansible earlier in the book and worked through some examples to see how powerful it is. When we worked with that earlier, I'm sure you immediately saw the benefit—not having to build a solution manually is a beautiful thing.

The importance of not having to build solutions manually cannot be overstated. Perhaps the most obvious benefit is the fact that it can save you hours, or even days of work. When I first started working in IT, setting up servers was always a manual task. Sure, you could create a Bash script and automate some tasks that way, but tools specifically designed to automate will handle the task much more efficiently. An IT staff that would normally be overwhelmed at the thought of setting up a large number of servers would be able to perform the same task much quicker with automation. And with all the time that's saved, IT staff members can focus on other tasks rather than spending the majority of their time on one task.

Another benefit of automation is that the likelihood of human error is much lower. While you're building your automation solution, making mistakes is unavoidable. You may mistype something while writing a script that causes a syntax error, or perhaps something doesn't get created quite the way you expected. But after you've spent the time building your automation scripts and verified there are no errors, then you can run them again and again and the infrastructure will get created the same way each time. Compare that to having to manually set up servers each time you wish to implement a new solution, and you can imagine how often it may happen that there may be mistakes to fix. Some of which you may not even discover until later on.

Automation also has another benefit you may not expect, **Disaster Recovery**. While we will cover disaster recovery in *Chapter 23*, *Preventing Disasters*, it's worth mentioning now because an effective automation solution will make the process of recovery quicker. It's an administrator's worst nightmare to even think about a server that's important to your organization may someday fail, but it's a fact of life.

Our organization may have a very complex application that consists of one or more web servers, a load balancer, security settings, and more. It could take hours to rebuild a solution like that manually. But with automation, you would simply run your scripts to recreate the same solution in mere minutes. Automation itself won't protect you from losing data (which would be an even scarier problem) but at the very least it can help you to provision replacement resources quicker than if you had to do the same manually. Not only that, I presume your clients (as well as your boss) will prefer your organization's application coming back online in minutes, rather than hours or days.

In addition, your automation scripts can serve as a form of living blueprint. Even if you aren't planning on re-provisioning your servers and related infrastructure, another administrator can look at your automation scripts and understand better which components make up the overall solution, allowing them to get up to speed quicker if they're taking over the management of infrastructure from someone else.

Automation is one of those things that I probably won't have to try too hard to sell you, because if you already have experience working in IT, then you already know how tedious it can be to manually rebuild servers. Sometimes, it may feel as though we have more tasks to complete than we have hours in the workday. But with automation, we can get some of that time back and possibly even lower our stress level a bit. And it's not the first time we've worked with automation; we did take a look at Ansible earlier in the book, so you are probably well aware of the benefits. But what we're going to do in this chapter is implement automation at a lower level than Ansible, and we'll do so with a solution known as **Terraform**. What is Terraform, you may ask? In science, terraforming is an amazing process of taking a planet that is uninhabitable and converting it into one that is able to support life as we know it. But for our purposes, Terraform is the name of an awesome utility we can use to automate an entire cloud computing implementation. In the next section, we'll define it even more.

## Introduction to Terraform and how it can fit within your workflow

Terraform is an amazing tool created by a company called Hashicorp that can automate your infrastructure at a level lower than Ansible, Puppet, or other configuration management solutions. In fact, Terraform typically doesn't replace those but complements them. With configuration management tools, we generally have to create the initial server and set up the operating system first before we can implement them. With Ansible, there are actually methods of using it to create infrastructure components, but that's beyond the scope of the book.

Not only that, but while Ansible is able to create some types of infrastructure, that's not what it does best. To understand where something like Terraform fits, it's best to think of Terraform as making things exist and Ansible as taking things that already exist and ensuring they're configured properly.

When it comes to Terraform itself, it allows you to take advantage of a neat concept, **Infrastructure as Code**. In the previous chapter, we set up an entire load-balanced application in AWS. We created an EC2 instance, as well as an AMI, and then we built the load balancer along with Auto Scaling. While that process was incredibly fun, it was a manual one. If you made mistakes during the process, you had to go and fix them. After you were done, your solution was created and working. What Terraform allows us to do is write code that represents our desired end state. When it runs, it checks the cloud provider and performs an inventory. If something we've added to our scripts isn't present with the cloud provider, it will make sure that the current state matches the desired end state in our code. We can even provision an entire cloud solution without even logging in to AWS beyond the first time.

An important consideration when it comes to automation tools is whether or not the tool is **cross-platform**. Many cloud providers feature built-in tools to do the same thing that Terraform does. For example, AWS has a feature called **CloudFormation** that allows you to script infrastructure builds, just as you can with Terraform. But the problem is that CloudFormation is specific to AWS. You can't utilize that service to build infrastructure in Microsoft Azure or Google Cloud. A tool that's cross-platform can run in any environment. We already saw this with Ansible earlier in the book: Ansible doesn't mind if the servers you're having it configure reside in AWS or even if they're physical machines in a rack. To Ansible, Ubuntu is Ubuntu, regardless of where it's running. This allows you to use the same tool in multiple environments, without having to recreate a new set of automation scripts for each one. Terraform is also a cross-platform tool.

Why does it matter if a tool is cross-platform? If you have to maintain several completely different tools that all do the same thing, it's a waste of time. If you can learn one tool and use it in every environment you support, then it's less of a maintenance burden. This is why I always recommend avoiding platform-specific tools, such as CloudFormation in AWS. There's even a tool within AWS called **OpsWorks** that's used for the same purpose as Ansible (configuration management), but again is specific to AWS.

A typical organization will pivot into different directions multiple times throughout the life of the company. An organization that is using AWS for 100% of its infrastructure may someday decide to support other cloud providers. Sometimes, all it takes is the right client or situation to make the company consider using a cloud provider for a project that would normally not be considered.

It could also be the case that a company might change primary providers due to a change made with the current platform that increases cost, or some other reason. If you use cross-platform tools, then you can take those tools with you (for the most part) if you change providers. Also, being able to support multiple providers not only makes you a more powerful administrator, but it also offers additional value to your organization.

Terraform itself is not going to be 100% identical between cloud platforms, though. The syntax does change from one cloud provider to another. Currently, there doesn't seem to be a solution available for infrastructure as code that is 100% portable between environments. But considering solutions such as CloudFormation are 0% transferable to other platforms, then Terraform still wins out in comparison since it is a tool you can use with multiple providers. The general consensus of how Terraform works will remain the same with each provider, so it's going to still save you time if you use it and then switch providers.

How does Terraform work? We'll install it in the next section, and actually use it to deploy an EC2 instance in the section after that. But in a nutshell, Terraform is a utility you can download to your local laptop or desktop, and use it to turn script files into actual infrastructure. It supports many different cloud platforms, such as AWS, GCP, and others. It even supports VPS providers, such as Digital Ocean and Linode. Terraform refers to each of those platforms as a provider and gives you the ability to download the appropriate plugin within Terraform to support your chosen provider(s).

As you'll see later in the chapter, Terraform allows you to test your configuration first, and preview the changes it will make. Then, if you accept the changes, it will connect to your provider and create the infrastructure as you've defined it in your code. Although we won't cover version control in this chapter, typical organizations will store their Terraform code within a Git repository or some other version control system, so that the code is safe from being accidentally deleted. In a typical organization, one or more administrators will work with the Terraform code, and push their changes into the repository.

In the next section, we'll walk through the process of installing Terraform so we will have everything we need in order to get started and build some automation around our infrastructure.

## Installing Terraform

The process of running Terraform and using it to provision your cloud resources is generally initiated on your local laptop or desktop. Terraform itself is downloaded from its website, and it's available for all of the leading operating systems.

Unlike the majority of applications, there's no installer. Terraform is run directly from the file you download, there's no installation process to go through. You can install it system-wide if you want to do so, but you can run it from any directory you wish. Download files for Terraform are located at the following website: `https://www.terraform.io/`.

Once there, you should see a download button labeled **Download CLI**:

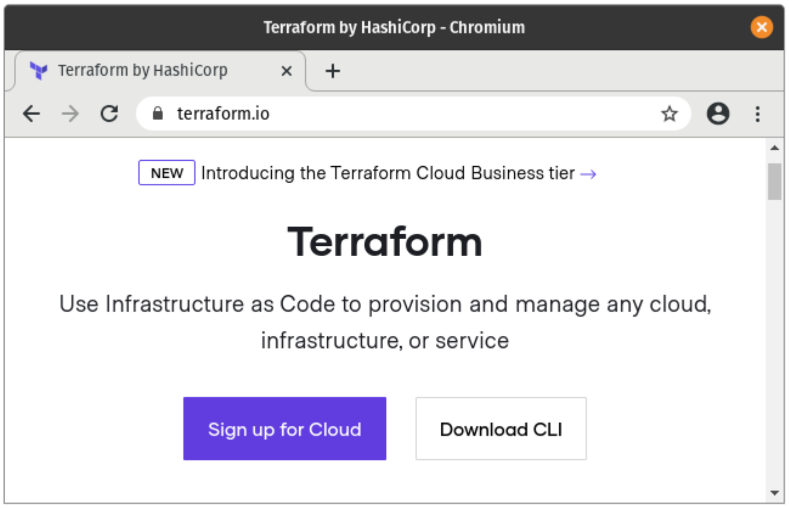

Terraform by HashiCorp - Chromium
Terraform by HashiCorp
terraform.io
NEW Introducing the Terraform Cloud Business tier →
Terraform
Use Infrastructure as Code to provision and manage any cloud, infrastructure, or service
Sign up for Cloud
Download CLI

Figure 20.1: The Terraform website

It's possible that the website could change at some point in the future, and the layout might be different. The download file for Terraform should be easy to find on the website even if the button moves around a bit. The CLI version is the one we'll use; the cloud version of Terraform is beyond the scope of this chapter. The main difference is that the CLI version allows you to initiate Terraform locally, whereas the cloud version is a hosted resource you can log into and utilize to provision your infrastructure right from within your web browser. If you or your organization intends to utilize Terraform heavily, the cloud version might be something worth considering.

When you proceed through the site to find the download file, you'll have quite a few different options available:

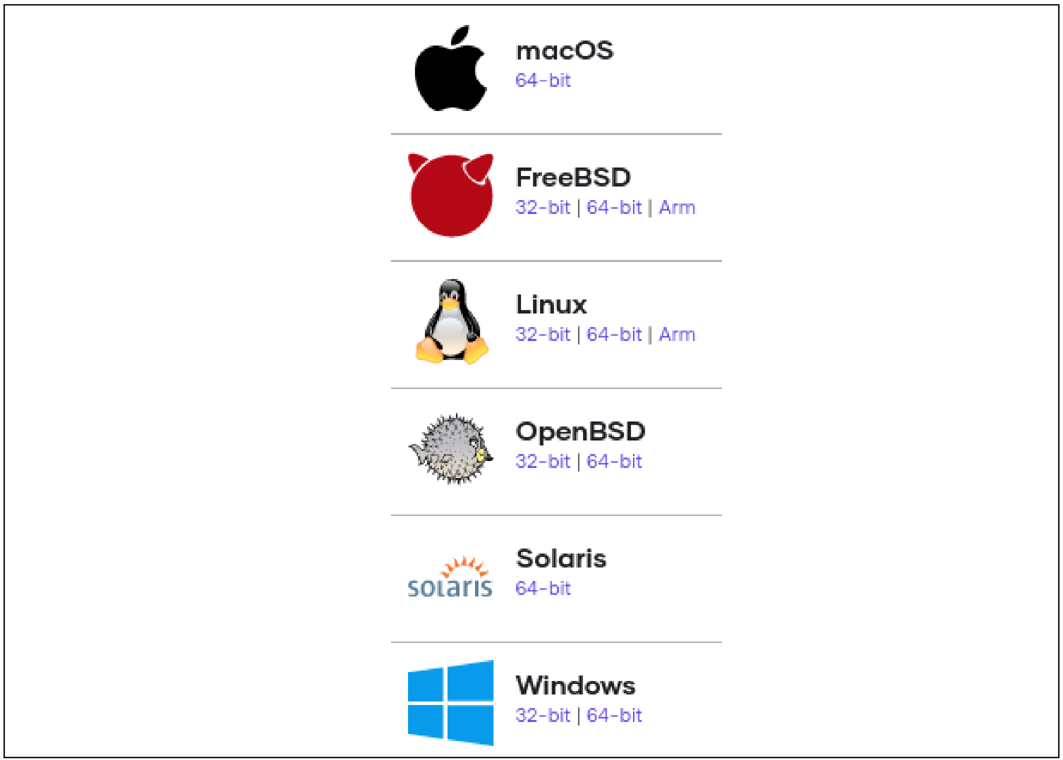
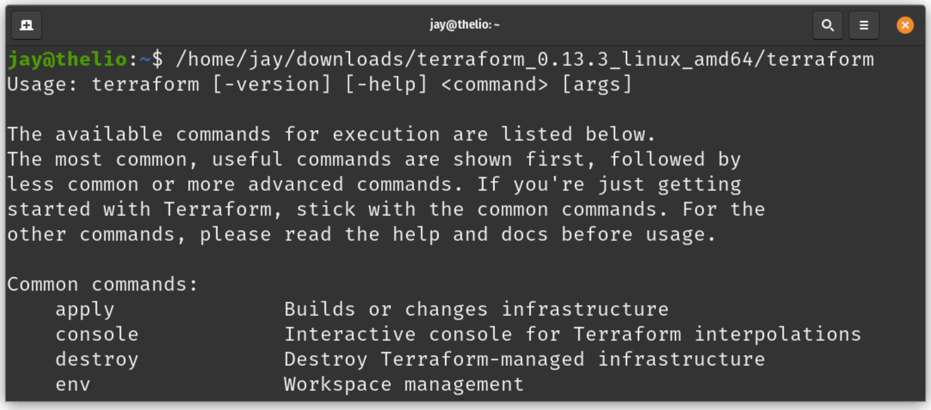


Figure 20.2: Download choices for Terraform

At this point, all you'll have to do is download a version of Terraform specific to your operating system. Most computers sold nowadays are 64-bit, so it should be straightforward to choose which one to download. If you'd like to run Terraform from a Raspberry Pi, choose the Arm version for Linux. Once you download it, you'll have a ZIP file locally that you can extract. Inside, you'll find a binary file simply titled `terraform` and that's all you'll actually need.

You'll be able to run `terraform` right from the command prompt of your operating system:



jay@thelio: ~

```
jay@thelio:~$ /home/jay/downloads/terraform_0.13.3_linux_amd64/terraform
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
```

Figure 20.3: Running terraform from a terminal window with no options

In the screenshot, I typed out the entire path to the downloaded and extracted `terraform` file, which was saved in my home directory under the `downloads` folder. I ran it with no options, so it printed out the help page.

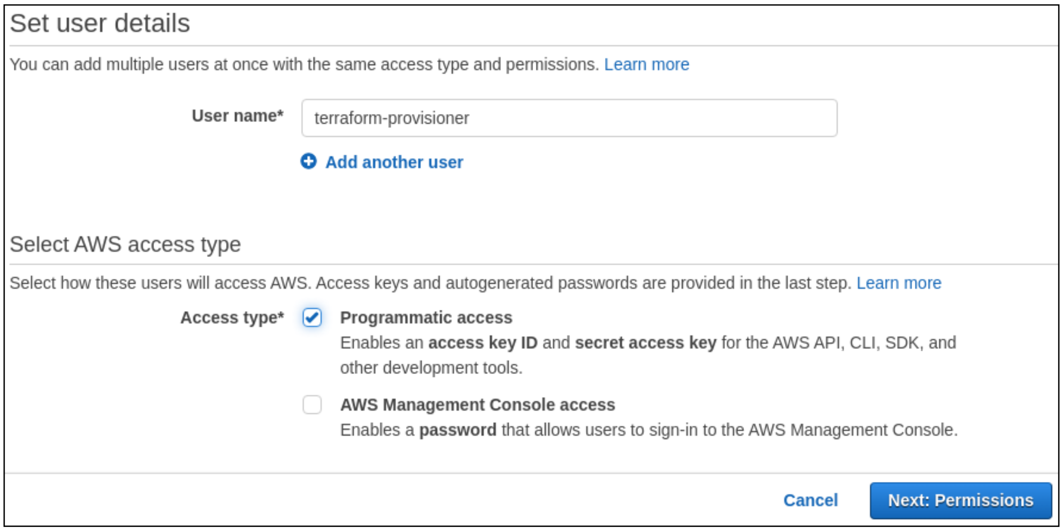
If you'd like to install it system-wide, you can move `terraform` into the `/usr/local/bin` directory if you're running Linux or macOS:

```
sudo mv terraform /usr/local/bin
```

The `/usr/local/bin` directory is recognized by both Linux and macOS as a directory that is searched for binary files. This concept is referred to as your `$PATH`, which is a special variable that holds all the directories your profile is set to look into when attempting to execute a command. The method of adding a new directory to your `$PATH` differs from one operating system to another, but in terms of macOS and Linux, `/usr/local/bin` is already recognized, so when you copy `terraform` into that directory, you should be able to simply type `terraform` in your terminal without needing to type the full path each time you wish to use Terraform. This is optional, but it makes it simpler.

In order for Terraform to be able to work with AWS, we'll need to generate an API key for it. This is done inside the AWS Management Console, which you should sign into now so we can create what we need. In the previous chapter, we discussed IAM, which is a service within AWS that allows you to not only create user accounts for fellow administrators, but it also lets you create keys for programmatic access. The latter will be how we allow Terraform to connect to our AWS account in order to perform tasks on our behalf.

Inside the IAM section of the AWS management console, click on the **Users** link that you should see in the left-hand menu, followed by the blue **Add User** button that you should see on that page. The following form will appear:


Set user details

You can add multiple users at once with the same access type and permissions. Learn more

User name*: terraform-provisioner

Select AWS access type

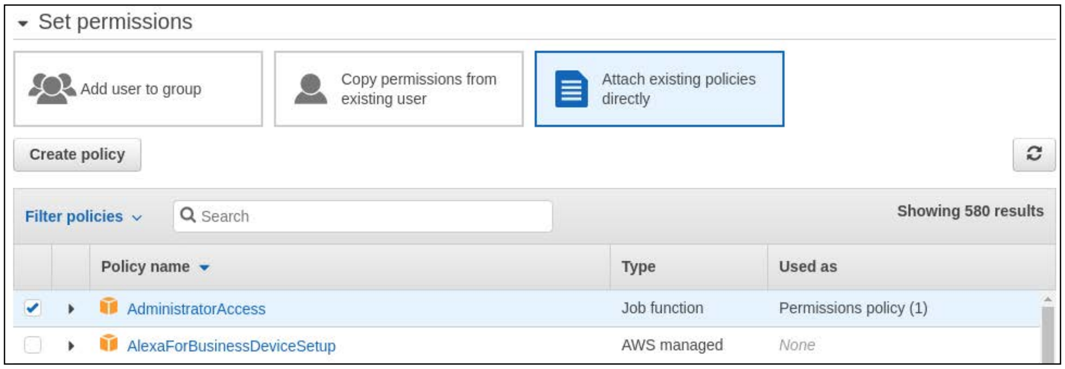
Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. Learn more

Access type*: [✓] Programmatic access — Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

[ ] AWS Management Console access — Enables a **password** that allows users to sign-in to the AWS Management Console.

Figure 20.4: Creating an IAM user for the purpose of running Terraform

In my case, I decided to call my user `terraform-provisioner`, but you can use whatever name you'd like. I checked the box next to **Programmatic access** and I left the second unchecked, because I do not want this user to be able to log in to the console. Click **Next: Permissions** to continue. On the next screen that comes up, we'll set the policy that the user will have access to:


Set permissions

Showing 580 results

| | Policy name | Type | Used as |
|---|---|---|---|
| ✓ | AdministratorAccess | Job function | Permissions policy (1) |
| | AlexaForBusinessDeviceSetup | AWS managed | None |

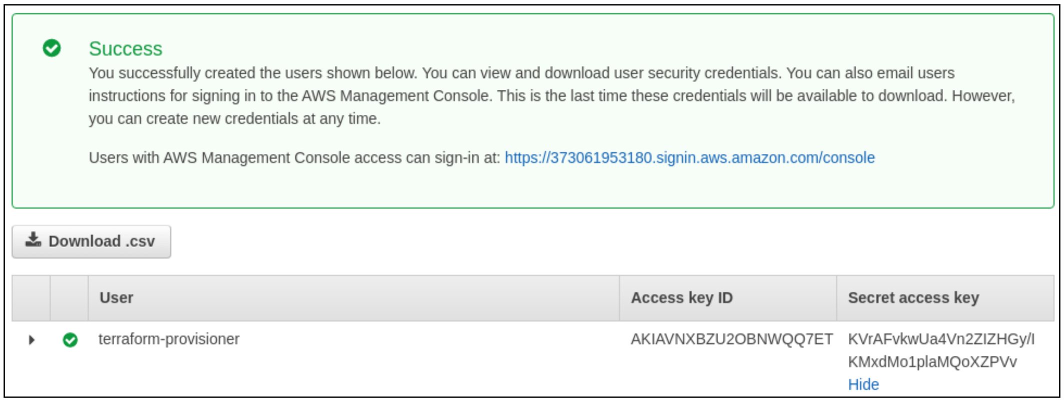
Add user to group
Copy permissions from existing user
Attach existing policies directly
Create policy
Filter policies
Search

Figure 20.5: Creating an IAM user for Terraform (continued)

For this screen, click on the box that reads **Attach existing policies directly** to highlight it, and check the box below to add the **AdministratorAccess** policy to this object. This is the policy that will grant Terraform its ability to interact with AWS.

Click **Next: Tags** to continue, then on the next screen you can skip adding tags (unless you'd like to add them) and you can click **Next: Review** and then **Create User** to finish the process.

The final screen that appears should report that the process was successful, and the **Download .csv** button gives you the ability to download your key. You can also reveal the secret access key by clicking the **Show** button, as I've done in *Figure 20.6*:



Success

You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: https://373061953180.signin.aws.amazon.com/console

| | | User | Access key ID | Secret access key |
|---|---|---|---|---|
| ▸ | ✓ | terraform-provisioner | AKIAVNXBZU2OBNWQQ7ET | KVrAFvkwUa4Vn2ZIZHGy/IKMxdMo1plaMQoXZPVv Hide |

Figure 20.6: Creating an IAM user for Terraform (final screen)

I'd like to give you a few warnings about the key, though. First, whether you download the key or reveal it by clicking the **Show** button, this is the last time you'll ever see it. You won't be given another opportunity to download the full key. I recommend you download the key and store it in a safe place. You should protect the key and not let anyone have access to it, and you should definitely not upload it to a version control repository or any other resource that's publicly available. And you should absolutely not show the key in clear text in a book that a bunch of people are going to read. If this key falls into the wrong hands, then anyone that has it will be able to interact with your AWS account. Treat this key with care. The only reason I show mine here is because I want you to see what the process actually looks like. I'll delete it from my AWS account before the publishing process of this book is finalized. On your end, definitely don't let this key leak.

Now we have everything we need to proceed to build AWS resources with Terraform. In the next section, we'll create an EC2 instance.

## Automating an EC2 instance deployment

Let's take a look at an example Terraform configuration file that will allow us to build an EC2 instance:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "my-server-1" {
  ami                                = "ami-0dba2cb6798deb6d8"
  associate_public_ip_address = "true"
  instance_type                      = "t2.micro"
  key_name                           = "jay_ssh"
  vpc_security_group_ids             = [ "sg-0597d57383be308b0" ]

  tags = {
    Name = "Web Server 1"
  }
}
```

Terraform files are saved with a `.tf` filename extension, and as for the actual name, you can call it whatever you wish. I named mine `terraform_example_1.tf`. The underscores in the filename aren't required but make it easier to use on the command line since you won't have to escape spaces. I placed my `terraform_example_1.tf` file inside a directory of its own, which is recommended. Your Terraform configuration files should be separate from other files, so having a dedicated directory for such files is ideal.

As for the actual code itself, lets explore it section by section:

```
provider "aws" {
  region = "us-east-1"
}
```

The `provider` block tells Terraform what type of provider we'll be working with. We're setting that to `aws` here. As mentioned earlier, Terraform is able to work with various cloud providers, of which AWS is only one. Underneath that, we're setting the `region` variable to `us-east-1`. On your end, I recommend setting this to whatever region you were using in the previous chapter; that will make the process easier for us since we already have some resources there that we can reuse for now.

```
resource "aws_instance" "my-server-1" {
```

Here, we're starting a new resource block. Each provider has their own building blocks (resources), and specific to AWS, we can use `aws_instance`. On this line, we're also naming the instance, and calling it `my-server-1`.

Note that this is a name within Terraform we're providing it, not the actual name that will be used in AWS itself. Within Terraform, we'll want to have some sort of name to refer back to this particular AWS instance if we need to refer to it again elsewhere.

```
    ami                         = "ami-0dba2cb6798deb6d8"
```

Next, we're choosing the AMI we'd like to use for our instance. As discussed in the previous chapter, an AMI is an image we can use to build a server in AWS. The instance ID that I used here is for the official Ubuntu 20.04 AMI that comes as default with AWS. AMIs are specific to the region they were created in, so the instance ID here is specific to `us-east-1`. If you're also using `us-east-1`, you can use the above AMI ID as-is (so long as it's not replaced by AWS with a newer one in the future). If in doubt, you can go into the AWS console, then navigate to the EC2 console, and go through the process as if you were going to manually create an EC2 instance based on Ubuntu, and copy the AMI ID from there. Perhaps even easier, you can use the Amazon EC2 AMI Locator (provided directly from Canonical) to find an AMI ID to use: `https://cloud-images.ubuntu.com/locator/ec2/`.

You're able to filter that list by Ubuntu version as well as location. That way, you can find the AMI ID for an Ubuntu 20.04 AMI that's within your chosen region. Change the AMI ID in the example code to the one you wish to use.

Also, you'll probably notice that there are quite a few spaces in between `ami` and `= "ami-0dba2cb6798deb6d8"`. It's common practice with Terraform syntax to align the equal sign of every line within a block, which makes the code look cleaner. This isn't required, and nothing bad will happen if you don't line everything up perfectly, but some may argue that the overall script looks cleaner that way.

```
    associate_public_ip_address = "true"
```

With this line, we're deciding to utilize a public IP address with our instance, which is required if we wish to be able to access it remotely.

```
    instance_type                           = "t2.micro"
```

Here, we're setting the desired instance type for our newly created server. As discussed in the previous chapter, there are multiple instance types available, each with a different cost. The `t2.micro` instance type is eligible for the free tier, so that's the reason I chose it.

```
    key_name                                 = "jay_ssh"
```

In the previous chapter, when we created an EC2 instance manually, part of that process was creating an OpenSSH key. The key that you've created is registered to your AWS account, so you can refer to it by the name you gave it. I called mine `jay_ssh`, but you'll want to change this to whatever you named yours. You can see a list of your OpenSSH keys in the EC2 dashboard within AWS, there's a section in the menu called **Key Pairs** where you can remind yourself what you've named your key if you forgot.

```
  vpc_security_group_ids        = [ "sg-0597d57383be308b0" ]
```

During the last chapter, we created a security group that allowed both Apache and OpenSSH to communicate with our instance. When you create a security group, it's designated with its own security group ID. The security group ID I used in the example was the one that was generated for me, so it won't work for you. If you access the **Security Groups** section of the EC2 console, you can find the security group ID for the one that you've created. The ID for it should start with `sg-`, followed by a series of characters. Add yours in place of what I have for mine in the example.

```
  tags = {
    Name = "Web Server 1"
  }
```

In the last section of the example, we're setting a tag. As discussed in the previous chapter, AWS allows you to create tags that are useful information you can have attached to an instance, which can give you additional information about its intended use. The `Name` tag is a special tag that changes the name that you see for the instance in the AWS EC2 list. You can name yours whatever you'd like.

At this point, we should be all set to go ahead and run Terraform to create our instance using our Terraform file as a blueprint. First, we need to set the Access and Secret Access keys for Terraform to use. In your terminal, you can enter the following commands to do this:

```
export AWS_ACCESS_KEY_ID="AKIAVNXBZU2OBNWQQ7ET"
export AWS_SECRET_ACCESS_KEY="KVrAFvkwUa4Vn2ZIZHGy/IKMxdMo1plaMQoXZPVv"
```

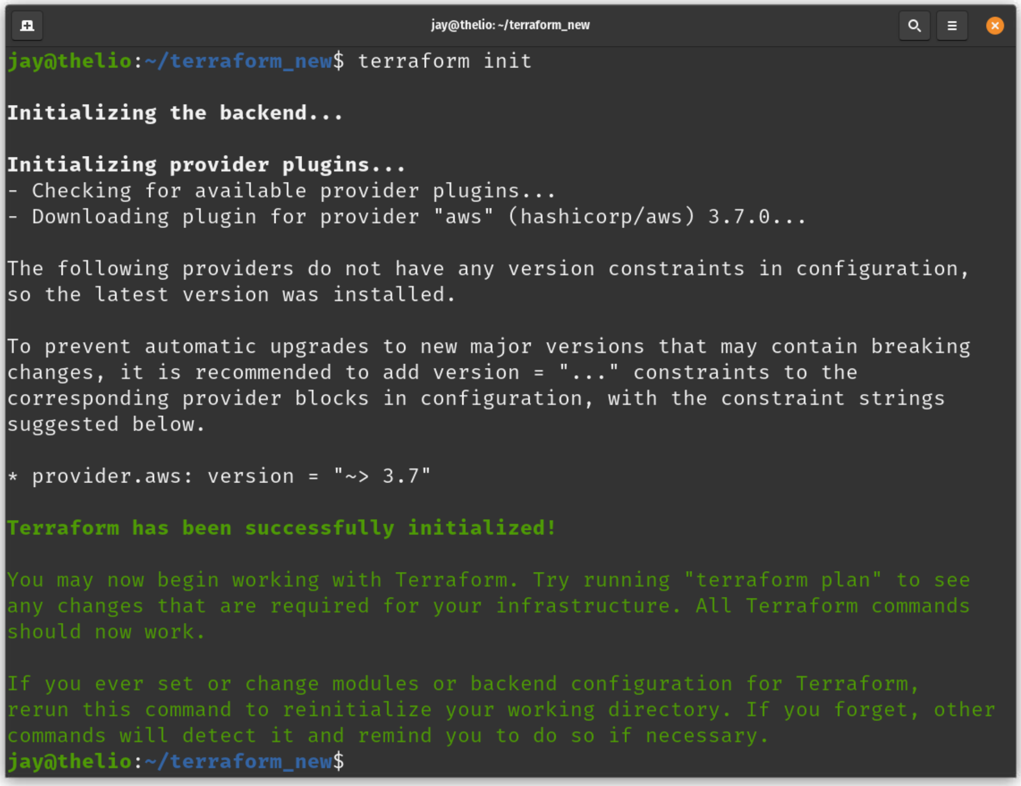
Those commands are simply run from your terminal and create environment variables containing the required keys. Terraform will look for the existence of the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` variables when it runs, and by exporting them, we're making them available in our session. There's actually a way you can add the keys right into the Terraform file itself, but we don't want to do that, because then the key might get uploaded somewhere public if we do upload the entire file somewhere. There's also a way to set up variables within the Terraform file to include these keys, but that's beyond the scope of this chapter.

After exporting the variables, we need to initialize Terraform to ensure it has the required components it needs to interact with AWS:

```
terraform init
```

With that example, you can add the full path to the `terraform` utility if it's not in a shared `$PATH` location, such as `/usr/local/bin` which is a recommended location that I mentioned earlier. If you did copy it to `/usr/local/bin`, then you should be able to simply type `terraform` instead of the full path.

The `terraform init` command instructs Terraform to initialize itself. It will look at any Terraform files you have in your current working directory and look for the `provider` line. In our case, that's at the beginning of the file. We set it to `aws`. This will trigger Terraform to download the provider addon for AWS:



```
jay@thelio:~/terraform_new$ terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 3.7.0...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 3.7"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
jay@thelio:~/terraform_new$
```

Figure 20.7: Initializing Terraform

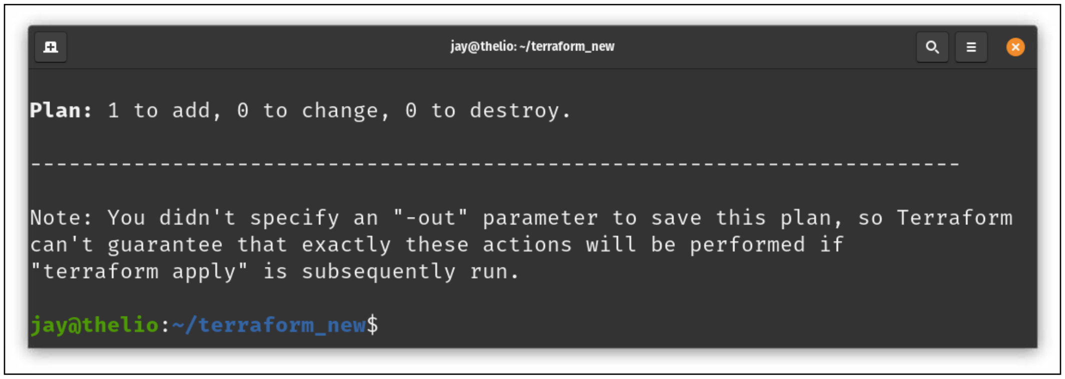
As you can see from *Figure 20.7*, when I ran the command on my end it downloaded the AWS provider to prepare it for use.

Next, we should run what's known as a **Terraform plan**. Running a plan instructs Terraform to *not* make any changes but instead to check your syntax and ensure that you haven't mistyped anything:

```
terraform plan
```

Terraform will connect to your provider, in our case AWS, to do an inventory and compare the changes in the configuration file to the current state of the provider. If it's unable to connect to the provider, an error will be returned. If the connection is successful, Terraform will list all the changes it would've made if you instructed it to actually perform the tasks. In plan mode, it will never actually carry out any instructions but merely provide you with a preview.

If for some reason Terraform can't connect to your AWS account, you should make sure you've run the two `export` commands earlier, and that you've done so with the appropriate values. If you close your terminal window, you'll need to run those `export` commands again since those environment variables do not persist between terminal sessions. If successful, the Terraform plan will run:



```
Plan: 1 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

jay@thelio:~/terraform_new$
```

Figure 20.8: Running a Terraform plan

In *Figure 20.8*, I've left off quite a bit of output. If your plan run was successful, Terraform will provide you with an overview of all the changes Terraform would've made if you were actually telling it to provision infrastructure.

If you would like to actually perform the changes, you can run a Terraform `apply` command. Before you do that though, always make it a habit to look at the output of the plan first. Notice the following line in the output:

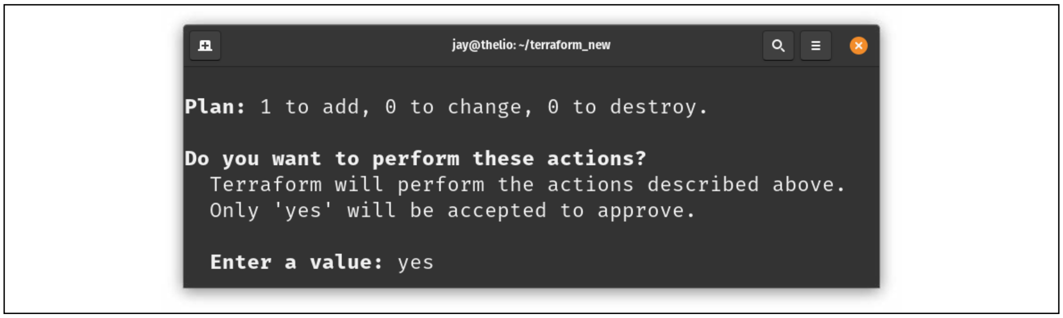
```
Plan: 1 to add, 0 to change, 0 to destroy
```

In our case, it's not going to destroy or change anything, but it's going to add something if it were to run. If you scroll up, you can find additional detail about the changes it might make if we were to run an `apply`. Pay special attention to what it might want to destroy. For some changes, Terraform may deem it necessary to delete something and recreate it from scratch. If you're using Terraform to update an existing server, you most likely won't want that server to be deleted. In that case, don't continue and run an `apply`. Always scrutinize the changes Terraform wants to make *before* you proceed and have it actually perform tasks.

Next, assuming we're comfortable with the changes, we'll proceed with an `apply`. Keep in mind that although running a plan with Terraform will cause it to look for and report syntax errors, passing the plan process with no syntax errors reported doesn't mean that there aren't any. There's only so much Terraform can do before you actually run it, so there may be errors that it can only catch during an `apply`. As you can guess, the command to run the `apply` is fairly obvious:

```
terraform apply
```

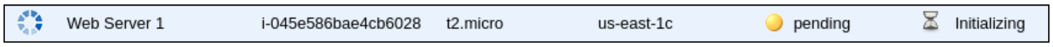
When it runs, the `terraform apply` command will run another sanity check, show the number of changes again, and ask if you'd like to continue:



```
Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

Figure 20.9: Running the terraform apply command

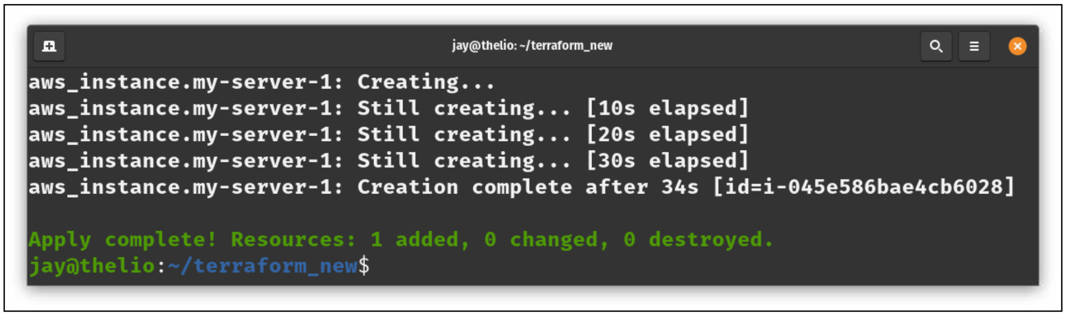
To proceed, type `yes` and press *Enter*. As it runs, you can actually see the resources you're having Terraform create show up in the AWS console as they're being provisioned. In the case of an EC2 instance as we're doing here, we'll be able to see the new instance in the list as it comes up:



| | | | | |
|---|---|---|---|---|
| Web Server 1 | i-045e586bae4cb6028 | t2.micro | us-east-1c | ● pending | Initializing |

Figure 20.10: A instance showing up in the EC2 list in AWS, created by Terraform

If all goes well, Terraform itself will report the process as being successful:



```
aws_instance.my-server-1: Creating...
aws_instance.my-server-1: Still creating... [10s elapsed]
aws_instance.my-server-1: Still creating... [20s elapsed]
aws_instance.my-server-1: Still creating... [30s elapsed]
aws_instance.my-server-1: Creation complete after 34s [id=i-045e586bae4cb6028]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
jay@thelio:~/terraform_new$
```

Figure 20.11: A successful terraform apply

Now we've created an EC2 instance in AWS, and we did so by utilizing automation. Sure, it hasn't done all that much for us yet, but this is just a proof of concept. There are many things we can do with Terraform.

There's something missing, though—security! We should also automate the process of adding a security group to the instance, which will provide us with the access we need to be able to connect to it and manage it. We're able to access the instance now, but it's very possible that it doesn't have external internet access yet. In the next section, we'll configure the security group for the instance as well, which will allow us to configure which ports are open and which IP addresses are able to communicate with our instance.

## Managing security groups with Terraform

Security groups, as you learned in the previous chapter, allow you to control what is able to communicate with your resources. In the previous section, we reused the security group that we've created last time, but it would be useful to understand how to create one from scratch.

Here's the example Terraform file again, with some new code added:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "my-server-1" {
  ami                                   = "ami-0dba2cb6798deb6d8"
  associate_public_ip_address = "true"
  instance_type                         = "t2.micro"
  key_name                              = "jay_ssh"
  vpc_security_group_ids        = [
  "${aws_security_group.external_access.id}" ]

  tags = {
    Name = "Web Server 1"
  }
}

resource "aws_security_group" "external_access" {
  name          = "my_sg"
  description = "Allow OpenSSH and Apache"

  ingress {
    from_port   = 22
    to_port        = 22
    protocol       = "tcp"
    cidr_blocks  = [ "172.11.59.105/32" ]
    description   = "Home Office IP"
  }

  ingress {
    from_port   = 80
    to_port        = 80
    protocol       = "tcp"
    cidr_blocks  = [ "172.11.59.105/32" ]
    description   = "Home Office IP"
  }
  egress {
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

I've added an entirely new section to the file, but before we get to that, I also changed a line from the previous example. It's the tenth line down:

```
vpc_security_group_ids        = [ "${aws_security_group.external_
access.id}" ]
```

Previously, we set the security group ID for this line in the file to the security group ID that already existed, the one we've created in the previous chapter. The configuration I've added further down will create a new security group, and here I'm setting the security group ID to a variable instead. The `${aws_security_group.external_access.id}` variable that was set here is known to Terraform as an **output variable**. We use an output variable for the security group ID, because we have no idea with the security group ID will be since the new security group hasn't even been created yet. Therefore, we use an output variable here, and reference the name of the security group we'll be creating (`external_access`) with `.id` at the end that will reference the security group ID once it's created. That way, we can reference a security group here and assign it to the instance, without having to know ahead of time what its ID will be.

Further down the file, we begin a new section:

```
resource "aws_security_group" "external_access" {
```

With that line, we're telling Terraform we'd like to create another new resource, this time a security group. We're giving this security group a name of `external_access`, which is just a name within Terraform we can reference it as, not an actual name it will be called within AWS.

```
  name          = "my_sg"
```

Here, we're giving the security group its actual name, the name we'll see it shown as within AWS outside of Terraform.

```
  description = "Allow OpenSSH and Apache"
```

For the `description` line, there's nothing too surprising here: we're giving it a description we can use to describe its purpose and what the security group will be used for. Similar to the security group we've created manually in the previous chapter, we'll be opening up OpenSSH and Apache with this security group.

```
  ingress {
    from_port   = 22
    to_port        = 22
    protocol       = "tcp"
    cidr_blocks  = [ "172.11.59.105/32" ]
    description   = "Home Office IP"
  }
```

The `ingress` block allows us to set a port to allow connections to come in from; in this case we're allowing connections from port `22`, which as you probably already know is the default port for OpenSSH. We don't want to open this port up to receive connections from the entire public internet, so we're allowing the incoming traffic to this port only if it's coming from an IP address of `172.11.59.105/32`. In your case, you can replace that with the public IP address of your home office or organization.

The second `ingress` block is the same as the first, only this time it's allowing connections to port `80` for Apache:

```
    ingress {
      from_port   = 80
      to_port        = 80
      protocol       = "tcp"
      cidr_blocks  = [ "172.11.59.105/32" ]
      description   = "Home Office IP"
    }
```

We also add an `egress` security group rule as well, as without this, our instance will not be able to reach the internet:

```
     egress {
      from_port = 0
      to_port = 0
      protocol = "-1"
      cidr_blocks = ["0.0.0.0/0"]
    }
```

As with the previous example, we'll need to run a plan and then an `apply` to transform our new code into reality. I'll leave it up to you to run both, as long as you haven't mistyped anything, it should apply the changes and add the new security group. Inside AWS, you should see the new security group in the console, and also see it applied to your EC2 instance. Unless you reused the AMI from the previous chapter with Apache built in, you won't be able to connect to the instance via port `80` since Apache is probably not installed, but I added it just to show you an example.

At this point, I recommend that you play around with the Terraform script we have so far, to get a feel for its syntax. Feel free to implement something extra, you can refer to the Terraform documentation for additional resources you can create with Terraform.

Congratulations on using Terraform to provision your infrastructure. Now, let's use Terraform to destroy stuff.

## Using Terraform to destroy unused resources

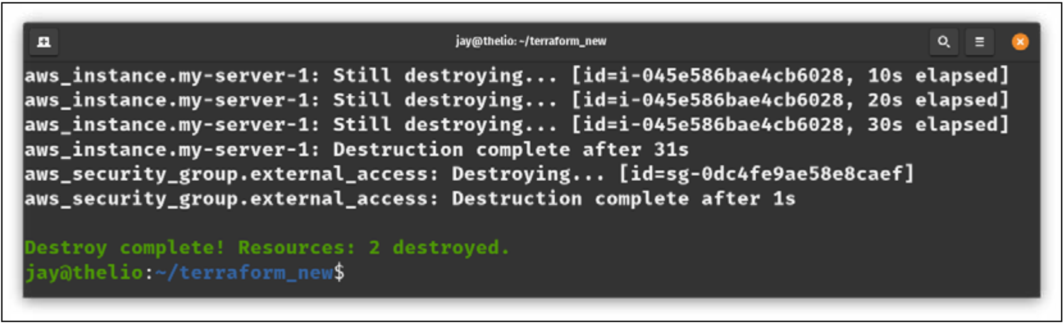
Although Terraform's primary purpose is to create infrastructure, it can also be used to delete infrastructure as well. This function is known as a **Terraform destroy**. With `destroy`, Terraform will attempt to remove all infrastructure that's defined in your configuration file. At this point, our configuration file creates an EC2 instance, as well as a security group. If we run `destroy` against it, then both resources will be removed.

Removing infrastructure with Terraform will likely be a use case you won't utilize as often as creating resources. One of the values of the `destroy` functionality, though, is that you can use it to "reset" a test environment, by removing everything defined in the file. Then you're free to use the same script to create everything again. On my end, I learn a lot faster by breaking things and fixing them. You really shouldn't run a `destroy` job against production infrastructure that you care about, but if you're just using Terraform in a test account that doesn't have any important instances inside it, then you can continually build and dismantle your test resources over and over as you learn. Another benefit is that an organization may test a Terraform build for a client in a test account first before implementing it in production, and you can verify that everything will be built correctly before performing the actual work for the client.

Performing a `destroy` within Terraform is just as simple as previous examples:

```
terraform destroy
```

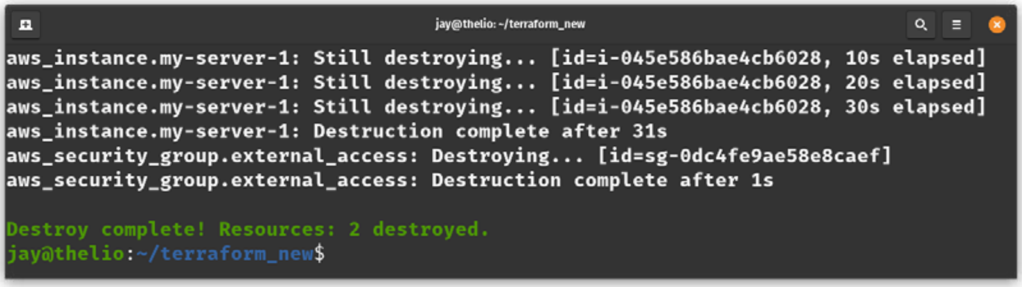
Just like before, we'll get a confirmation first before it removes everything, showing us exactly what Terraform wants to remove when a `destroy` task is run:



```
aws_instance.my-server-1: Still destroying... [id=i-045e586bae4cb6028, 10s elapsed]
aws_instance.my-server-1: Still destroying... [id=i-045e586bae4cb6028, 20s elapsed]
aws_instance.my-server-1: Still destroying... [id=i-045e586bae4cb6028, 30s elapsed]
aws_instance.my-server-1: Destruction complete after 31s
aws_security_group.external_access: Destroying... [id=sg-0dc4fe9ae58e8caef]
aws_security_group.external_access: Destruction complete after 1s

Destroy complete! Resources: 2 destroyed.
jay@thelio:~/terraform_new$
```

Figure 20.12: Preparing to run terraform destroy to remove resources

Pay careful attention to what Terraform wants to delete when you run it with the `destroy` option. The screenshot doesn't show the full output; it's quite long. Similar to `apply`, if you scroll up, you'll see that the output will contain detail about what in particular will be removed if we agree to continue. If you type `yes` and press *Enter*, the resources identified will be destroyed, and you'll receive a confirmation message confirming that the task was carried out:



```
aws_instance.my-server-1: Still destroying... [id=i-045e586bae4cb6028, 10s elapsed]
aws_instance.my-server-1: Still destroying... [id=i-045e586bae4cb6028, 20s elapsed]
aws_instance.my-server-1: Still destroying... [id=i-045e586bae4cb6028, 30s elapsed]
aws_instance.my-server-1: Destruction complete after 31s
aws_security_group.external_access: Destroying... [id=sg-0dc4fe9ae58e8caef]
aws_security_group.external_access: Destruction complete after 1s

Destroy complete! Resources: 2 destroyed.
jay@thelio:~/terraform_new$
```

Figure 20.13: Final confirmation after destroying previously provisioned resources

Basic usage of the `terraform` command is logically structured, we looked at how to run a `plan` as well as an `apply`, and now we now how to destroy our resources as well so we can start over with a clean slate. The majority of the time spent learning Terraform will be a matter of learning the syntax of its config files, but that will come in time. At this point in our journey, you should have a solid foundation we can build upon.

However, we're not done yet! I've referenced Ansible several times in this chapter, reminding you about the fact that we used it in the past to configure a server. But what if I told you we can combine Terraform and Ansible? We certainly can, and we'll do so in the next section.

## Combining Ansible with Terraform for a full deployment solution

One of the best things about automation tools is that they can often be combined to offer a shared benefit. Ansible is one of my favorite tools: you can automate the installation of packages, the creation of users, the copying of files, or most other tasks you can think of. If you are able to perform a task on the command line, chances are Ansible can automate it. Terraform, as you just saw, is really good at creating new infrastructure and automating the initial setup of servers, as well as networks and settings for AWS and other platforms. If we combine the two, it gets even better.

I find the duo of Terraform and Ansible to be a great fit. Combining these two solutions works well in my experience; we can use Terraform to create our initial server and infrastructure builds, and then use Ansible to automate future enhancements. But it's actually even better than that, we can configure Terraform to actually launch the initial Ansible run for us, so we only have to run a single script. After Terraform creates the infrastructure, provisioning of additional settings is handed off to Ansible. It's a great combination.

How does it work? In the previous chapter, we explored the concept of user data, which is a feature within AWS that allows you to run a script as an instance is being created. We used it to install all the patches and then proceed and install Apache. The example we went over was a simple Bash script and wasn't very exciting in and of itself. Sure, it did work, but we can implement a better solution. And you know what? We already have. In *Chapter 15*, *Automating Server Configuration with Ansible*, we were able to utilize Ansible Pull, a special mode of Ansible that allows us to pull code from a repository, and run it locally on our instance. The Ansible playbook we wrote installs Apache for us, the same as our Bash script did in the previous chapter. As a refresher, we are able to run the following command to trigger Ansible Pull:

```
ansible-pull -U https://github.com/myusername/ansible.git
```

Of course, this requires Ansible itself to be installed, and the repository needs to already exist. If you already followed along in that chapter and you still have the repository we've created, then you already have what you need to combine Ansible with Terraform. To save you the trouble of flipping back to *Chapter 15*, *Automating Server Configuration with Ansible*, here's the final `local.yml` file we ended with:

```
---

- hosts: localhost
  become: true
  tasks:

  - name: Install Apache
    apt: name=apache2

  - name: Start the apache2 services
    service:
      name: apache2
      state: started

  - name: Copy index.html
    copy:
      src: index.html
      dest: /var/www/html/index.html
```

As you can see, this playbook is installing Apache, starting it, and also copying an `index.html` file to replace the default web page. It's fairly easy to implement this in Terraform. Here's our Terraform script again, with a new line added, shown in bold:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "my-server-1" {
  ami                                   = "ami-0dba2cb6798deb6d8"
  associate_public_ip_address = "true"
  instance_type                         = "t2.micro"
  key_name                              = "jay_ssh"
  vpc_security_group_ids         = [ "${aws_security_group.external_
access.id}" ]
  user_data = file("bootstrap.sh")

  tags = {
    Name = "Web Server 1"
  }
}

resource "aws_security_group" "external_access" {
  name = "my_sg"
  description = "Allow OpenSSH and Apache"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = [ "173.10.59.105/32" ]
    description = "Home Office IP"
  }

  ingress {
    from_port   = 80

    to_port     = 80
```

```
      protocol    = "tcp"
      cidr_blocks = [ "173.10.59.105/32" ]
      description = "Home Office IP"
    }
    egress {
      from_port = 0
      to_port = 0
      protocol = "-1"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
```

The new addition to the file is on line #11. We're referencing a bootstrap script, and in that script, we'll add any commands we wish to run on the newly created instance:

```
    user_data = file("bootstrap.sh")
```

`bootstrap.sh` will need to exist in the same directory as the Terraform configuration file itself. The file doesn't exist yet though, so go ahead and create it, and inside you can place the following lines:

```
   #!/bin/bash
  sudo apt update
  sudo apt install -y ansible
  sudo ansible-pull -U https://github.com/myusername/ansible.git
```

We haven't made an overly complex change to the file, but what we did add gives us a great deal of benefit. The `user_data` option allows us to leverage the same user data function that's built into AWS and schedule commands to run when an instance is first created. In this example, we utilize the `user_data` option to run a series of commands against the new instance, which will install Ansible and then launch `ansible-pull` to download a repository containing an Ansible playbook and run it locally. The playbook itself was already set up in *Chapter 15, Automating Server Configuration with Ansible*, so we're just leveraging what we've already created in the past, and we're having Terraform kick off the Ansible job for us when it brings up the instance.

That brings us to the end of this chapter. I hope setting up automation with Terraform was a fun experience, I definitely enjoy working with it.

## Summary

There are many configuration management and provisioning tools available for automating our infrastructure builds. In this chapter, we took a look at Terraform, and then we even combined it with Ansible, which we were already using. Using Terraform, we were able to automate the creation of an EC2 instance in AWS, along with a security group to control how it can be accessed. Terraform itself is a very large subject, and the concepts contained in this chapter are only the beginning. There's so much more you can do with Terraform, and I highly recommend you keep practicing with it and coming up to speed.

In the next chapter, we're going to learn some methods we can utilize to add additional security to our Ubuntu servers. While no server is bulletproof, there's a basic level of security we can implement that will make it less likely for our server to be compromised. It will be a very important chapter, so you won't want to miss it.

# 21
# Securing Your Server

It seems like every month there are new reports about companies getting their servers compromised. In some cases, entire databases end up freely available on the internet, which may even include sensitive user information that can aid miscreants in stealing identities. Linux is a very secure platform, but it's only as secure as the administrator who sets it up. Security patches are made available on a regular basis, but they offer no value unless you install them. OpenSSH is indispensable for remote administration, but it's also a popular target for threat actors trying to break into servers. Backups are a must-have but are potentially useless if they're not tested regularly or they fall into the wrong hands. In some cases, even your own employees can cause intentional or unintentional damage. In this chapter, we'll look at some of the ways you can secure your servers from threats.

In this chapter, we will cover:

- Lowering your attack surface
- Understanding and responding to CVEs
- Installing security updates
- Automatically installing patches with the Canonical Livepatch service
- Monitoring Ubuntu servers with the Canonical Landscape service
- Securing OpenSSH
- Installing and configuring Fail2ban
- MariaDB best practices for secure database servers

- Setting up a firewall
- Encrypting and decrypting disks with LUKS
- Locking down `sudo`

To get started, let's first talk about ways you may be able to lower your attack surface.

## Lowering your attack surface

Your Ubuntu Server installations will likely have one or more important applications running on them, some of which might be available to the public internet. This is very common for web servers, for example, as it's the primary goal of a web server to offer a website that your users can access. Every application that is accessible from outside the walls of your organization is a potential entry point for threat actors who might attempt to break into your server. The **attack surface** of a server is essentially a list of all the things that are potentially exploitable. In regards to security, it's important to understand which applications must be accessible remotely, and which ones you can lock down. Every application you lock down lowers the likelihood of it being taken over by an outside threat. The process of locking things down is what we refer to as lowering your attack surface.

Ideally, in a perfect world, we would disallow all outside connections to all of our servers. Hackers can't break into a server that is completely inaccessible from the outside. That doesn't mean that there aren't any threats at all, as disgruntled employees are always a potential risk. But a server that's completely inaccessible is the most secure of all. However, it's often not feasible to disallow all outside connections. If your company provides a popular public website, then it has to be publicly available. However, if you have an application running on your server that is only used by users internally, then you should lock it down if you can. Whenever possible, it's good to implement a policy that outside connections are always disallowed by default unless there's a business need to open it up.

What do we mean by "disallow?" There are multiple ways you can disallow access to an application on your server. The most effective of which is to completely uninstall the application. If you don't have an application installed at all, it's impossible for it to be a problem. It probably goes without saying that you should uninstall applications that aren't necessary, but the entire point of running a server is to serve resources to users, so you'll always have applications running on your server (otherwise there wouldn't be a point in having a server at all to begin with). Aside from removing an application, you can utilize a firewall to only allow specific connections. We'll actually take a look at setting up a firewall later on in this chapter.

Most importantly, after a new server is implemented, an administrator should always perform a security check to ensure that it's as secure as it can possibly be. No administrator can think of everything, and even the best among us can make a mistake, but it's always important that we do our best to ensure we secure a server as much as we can. There are many ways you can secure a server, but the first thing you should do is lower your attack surface. This means that you should close as many holes as you can, and limit the number of things that outsiders could potentially be able to access. In a nutshell, if it's not required to be available from the outside, lock it down. If it's not necessary at all, remove it.

To start inspecting your attack surface, the first thing you should do is see which ports are listening for network connections. When an attacker wants to break into your server, it's almost certain that a port scan is the first thing they will perform. They'll inventory which ports on your server are listening for connections, determine what kind of application is listening on those ports, and then try a list of known vulnerabilities to try to gain access. To inspect which ports are listening on your server, you can do a simple port query with the `ss` command:

```
sudo ss -tulpn
```

The `sudo` portion of that command is optional, but if you do include `sudo`, you'll see more information in the output. Normally I'd include a screenshot here, but there's so much information that it won't fit on this page. From the output, you'll see a list of ports that are listening for connections. If the port is listening on `0.0.0.0`, then it's listening for connections from any network. This is bad. If the port is listening on `127.0.0.1`, then it's not actually accepting outside connections. Take a minute to inspect one of your servers with the `ss` command and note which services are listening for outside connections.

Armed with the knowledge of what ports your server is listening on, you can make a decision about what to do with each one. Some of those ports may be required, as the entire purpose of a server is to serve something, which usually means communicating over the network. All of these legitimate ports should be protected in some way, which usually means configuring the service after reviewing its documentation for best practices (which will depend on the particular service) or enabling a firewall, which we'll get to in the *Setting up a firewall* section. If any of the ports are not needed, you should close them down. You can either stop their daemon and disable it, or remove the package outright. I usually go for the latter, since it would just be a matter of reinstalling the package if I changed my mind.

OpenSSH is a service that you're almost always going to have running on your servers. As you are already well aware, it's a great tool for remote administration. But as useful as it is, it's usually going to be the first target for any attacker attempting to gain entry into your server.

We won't want to remove this though, because it's something we'll want to take advantage of. What should we do? Not to worry, I'll be dedicating a section to securing OpenSSH later in this chapter. I mention this now in order to make sure you're aware that lowering your attack surface will absolutely need to include at least a basic amount of security tweaking for OpenSSH. In addition, I'll go over Fail2ban in this chapter as well, which can help add an additional layer of security to OpenSSH.

As I've mentioned, I'm a big fan of removing packages that aren't needed. The more packages you have installed, the larger your attack surface is. It's important to remove anything you don't absolutely need. Even if a package isn't listed as an open port, it could still be leveraged in a **vulnerability chain**. If an attacker uses a vulnerability chain, that essentially means that they first break into one service and then use a vulnerability in another (possibly unrelated) package to elevate their privileges and attempt to gain full access. For that reason, I will need to underscore the fact that you should remove any packages you don't need on your server. An easy way to get a list of all the packages you have installed is with the following command:

```
dpkg --get-selections > installed_packages.txt
```

This command will result in the creation of a text file that will contain a list of all the packages that you have installed on your server. Take a moment to look at it. Does anything stand out that you know for sure you don't need? You most likely won't know the purpose of every single package, and there could be hundreds or more. A lot of the packages that will be contained in the text file are distribution-required packages you can't remove if you want your server to boot up the next time you go to restart it. If you don't know whether or not a package can be removed, do some research on Google. If you still don't know, maybe you should leave that package alone and move on to inspect others. By going through this exercise on your servers, you'll never really remember the purpose of every single package and library, but you should still find some things that you'll be able to clean up. Eventually, you'll come up with a list of typical packages most of your servers don't need, which you can make sure are removed each time you set up a new server. You could even curate a list of unneeded packages, and then create an Ansible playbook to make sure they're not installed.

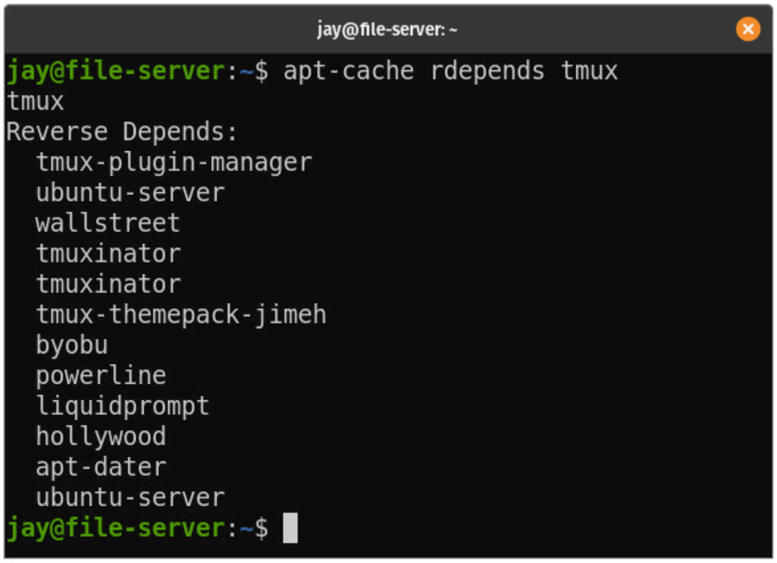
While attempting to clean up unneeded packages, a useful trick is to use the following command to check whether or not other packages depend on the package you are thinking of removing:

```
apt-cache rdepends <package-name>
```

As an example, I ran that command against the `tmux` package that I installed on a test server, but you can use whichever package name you'd like as an argument to check to see if anything depends on it:

```
apt-cache rdepends tmux
```

The output I received on my end is the following:



```
jay@file-server:~$ apt-cache rdepends tmux
tmux
Reverse Depends:
  tmux-plugin-manager
  ubuntu-server
  wallstreet
  tmuxinator
  tmuxinator
  tmux-themepack-jimeh
  byobu
  powerline
  liquidprompt
  hollywood
  apt-dater
  ubuntu-server
jay@file-server:~$
```

Figure 21.1: Updating packages on an Ubuntu server

With the output of the previous command, you can easily identify if another package depends on the package you are thinking about removing. In the example output, we can see that `tmux` is actually installed as a dependency of the `ubuntu-server` package. This means that `tmux` is quite possibly installed by default on your server, but that may vary depending on whether or not you've installed Ubuntu Server yourself or are using a cloud image. Cloud providers don't always configure Ubuntu Server images the same way. But at the very least, you can identify dependencies and make a more informed decision on whether or not you can safely remove a package.

Even if the output shows a package has no dependencies, you still may not want to remove it unless you understand the functionality it provides and what impact removing the package may have on your system. You can always Google the package name for more details, but at the very least you should look for open ports and focus on those first, since open ports have a greater impact on the security of your server. We'll look at this in more detail later in this chapter, in the *Setting up a firewall* section.

Another important consideration is making sure to use only strong passwords. This probably goes without saying, since I'm sure you already understand the importance of strong passwords. However, I've seen hacks recently in the news caused by administrators who set weak passwords for their external-facing database or web console, so you never know. The most important rule is that if you must have a service listening for outside connections, then it absolutely must have a strong, randomly generated password. Granted, some daemons don't have a password associated with them (Apache is one example; it doesn't require authentication for someone to view a web page on port `80`). However, if a daemon does have authentication, it should have a very strong password. OpenSSH is an example of this. If you must allow external access to OpenSSH, that user account should have a strong randomly generated password. Otherwise, it will likely be taken over within a couple of weeks by a multitude of bots that routinely go around scanning for these types of things. In fact, it's best to disable password authentication in OpenSSH entirely, which we will do later in this chapter. Disabling password authentication increases the security around OpenSSH quite a bit.

Finally, it's important to employ the **principle of least privilege** for all your user accounts. You've probably gotten the impression from several points I've made throughout the book that I distrust users. While I always want to think the best of everyone, sometimes the biggest threats can come from within (disgruntled employees, accidental deletions of critical files, and so on). Therefore, it's important to lock down user accounts as much as possible, and allow them access to only what they actually need to perform their job. This may involve, but is certainly not limited to:

- Adding a user to the smallest possible number of groups
- Defaulting all network shares to read-only (users can't delete what they don't have permission to delete)
- Routinely auditing all your servers for user accounts that have not been logged into for a time

- Setting account expirations for user accounts, and requiring users to re-apply to maintain account status (this prevents hanging user accounts)
- Allowing user accounts to access as few system directories as possible (preferably none, if you can help it)
- Restricting `sudo` to specific commands (more on that later on in this chapter)

Above all, make sure you document each of the changes you make to your servers, in the name of security. After you develop a good list, you can turn that list into a security checklist to serve as a baseline for securing your servers. Then, you can set reminders to routinely scan your servers for unused user accounts, unnecessary group memberships, and any newly opened ports.

Now you should have some good ideas on how you can lower your attack surface. It's also important to keep up to date on the current trends and notices surrounding security issues that were reported. In the next section, we'll take a look at **Common Vulnerabilities and Exposures** (**CVEs**), which can help you better understand the nature of threats in the wild.

## Understanding and responding to CVEs

I've already mentioned some of the things you can do in order to protect your server from some common threats, and I'll give you more tips later on in this chapter. But how does one know when there's a vulnerability that needs to be patched? How do you know when to take action? The best practices I'll mention in this chapter will only go so far; at some point, there may be some sort of security issue that will require you to do something beyond generating a strong password or locking down a port.

The most important thing to do is to keep up with the news. Subscribe to sites that report news on security vulnerabilities, and I'll even place a few of these in the *Further reading* section of this chapter. When a security flaw is revealed, it's typically reported on these sites, and given a CVE number where security researchers will document their findings.

CVEs are found in special online catalogs detailing security vulnerabilities and their related information. In fact, many Linux distributions (Ubuntu included) maintain their own CVE catalogs with vulnerabilities specific to their platform. On such a page, you can see which CVEs the version of your distribution is vulnerable to, have been responded to, and what updates to install in order to address them.

Often, when a security vulnerability is discovered, it will receive a CVE identification right away, even before mitigation techniques are known. In my case, I'll often watch a CVE page for a flaw when one is discovered, and look for it to be updated with information on how to mitigate it once that's determined.

Most often, closing the hole will involve installing a security update, which the security team for Ubuntu will create to address the flaw. In some cases, the new update will require restarting the server or at least a running service, which means I may have to wait for a maintenance period to perform the mitigation.

I recommend taking a look at the Ubuntu CVE tracker, available at `https://people.canonical.com/~ubuntu-security/cve/`. On this site, Canonical (the makers of Ubuntu) keep information regarding CVEs that affect the Ubuntu platform. There, you can get a list of vulnerabilities that are known to the platform as well as the steps required to address them. There's no one rule about securing your server, but paying attention to CVEs is a good place to start. We'll go over installing security updates in the next section, which is the most common method of mitigation.

## Installing security updates

Since I've mentioned updating packages several times, let's have a formal conversation about it. Updated packages are made available for Ubuntu quite often, sometimes even daily. These updates mainly include the latest security updates but may also include new features. Since Ubuntu 20.04 is an LTS release, security updates are much more common than feature updates. Installing the latest updates on your server is a very important practice, but, unfortunately, it's not something that all administrators keep up on for various reasons.

When installed, security updates very rarely make many changes to your server, other than helping to keep it secure against the latest threats. However, it's always possible that a security update that's intended to fix a security issue ends up breaking something else. This is rare, but I've seen it happen. When it comes to production servers, it's often difficult to keep them updated, since it may be catastrophic to your organization to introduce change within a server that's responsible for a large portion of your profits. If a server goes down, it could be very costly. Then again, if your servers become compromised and your organization ends up the subject of a CNN hacking story, you'll definitely wish you had kept your packages up to date!

The key to a happy data center is to test all updates before you install them. Many administrators will feature a system where updates will *graduate* from one environment into the next. For example, some may create virtual clones of their production servers, update them, and then see whether anything breaks. If nothing breaks, then those updates will be allowed on the production servers.

In a clustered environment, an administrator may just update one of the production servers, see how it gets impacted, and then schedule a time to update the rest. In the case of workstations, I've seen policies where select users are chosen for security updates before they are uploaded to the rest of the population. I'm not necessarily suggesting you treat your users as guinea pigs, but everyone's organization is different, and finding the right balance for installing updates is very important. Although these updates represent change, there's a reason that Ubuntu's developers went through the hassle of making them available. These updates fix issues, some of which are security concerns that are already being exploited as you read this.

To begin the process of installing security updates, the first step is to update your local repository index. As we've discussed before, the way to do so is to run `sudo apt update`. This will instruct your server to check all of its subscribed repositories to see whether any new packages were added or whether any out-of-date packages were removed. Then, you can start the actual process.

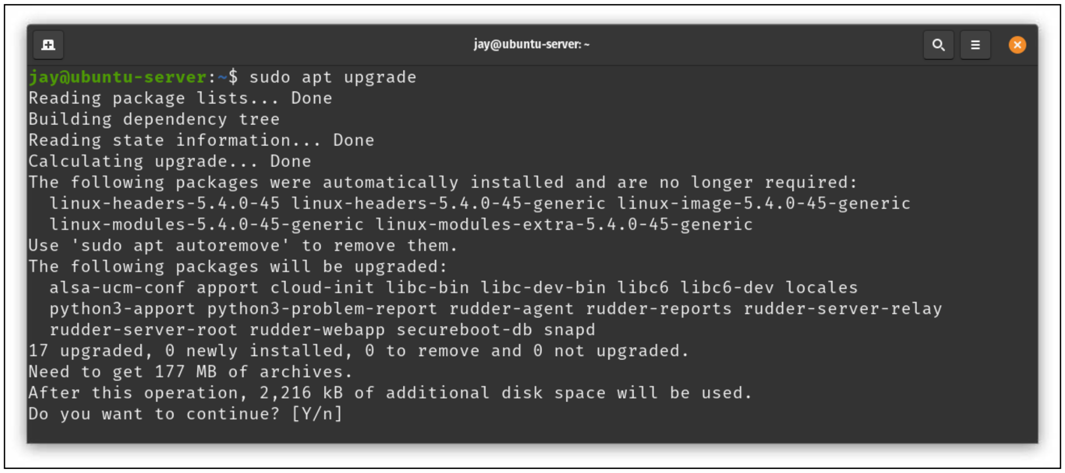
There are two commands you can use to update packages. You can run either `sudo apt upgrade` or `sudo apt dist-upgrade`.

The difference is that running `apt upgrade` will not remove any packages and is the safest to use. However, this command won't pull down any new dependencies either. Basically, the `apt upgrade` command simply updates any packages on your server that have already been installed, without adding or removing anything. Since this command won't install anything new, this also means your server will not have updated kernels installed either.

The `apt dist-upgrade` command will update absolutely everything available. It will make sure all packages on your server are updated, even if that means installing a new package as a dependency that wasn't required before. If a package needs to be removed in order to satisfy a dependency, it will do that as well. If an updated kernel is available, it will be installed. If you use this command, just take a moment to look at the proposed changes before you agree to have it run, as it will allow you to confirm the changes during the process.

Generally speaking, the `dist-upgrade` variation should represent your end goal, but it's not necessarily where you should start. Updated kernels are important, since your distribution's kernel receives security updates just like any other package. All packages should be updated eventually, even if that means something is removed because it's no longer needed or something new ends up getting installed.

When you start the process of updating, it will look similar to the following:



```
jay@ubuntu-server:~$ sudo apt upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
The following packages were automatically installed and are no longer required:
  linux-headers-5.4.0-45 linux-headers-5.4.0-45-generic linux-image-5.4.0-45-generic
  linux-modules-5.4.0-45-generic linux-modules-extra-5.4.0-45-generic
Use 'sudo apt autoremove' to remove them.
The following packages will be upgraded:
  alsa-ucm-conf apport cloud-init libc-bin libc-dev-bin libc6 libc6-dev locales
  python3-apport python3-problem-report rudder-agent rudder-reports rudder-server-relay
  rudder-server-root rudder-webapp secureboot-db snapd
17 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Need to get 177 MB of archives.
After this operation, 2,216 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Figure 21.2: Updating packages on an Ubuntu server

Before the update process actually starts, you'll be given an overview of what it wants to do. In my case, it wants to upgrade `17` packages. If you were to enter `Y` and then press *Enter*, the update process would begin. At this point, I'll need to leave the terminal window open; it's actually dangerous to close it in the middle of the update process. Closing the terminal window in the middle of a package management task may result in corrupted or partially installed packages.

Assuming that this process finishes successfully, we can run the `apt dist-upgrade` command to update the rest; specifically, the packages that were held back because they would've installed new packages or removed existing ones. There weren't any in my case, but in such a situation you may see text indicating that some upgrades were held back, which is normal with `apt upgrade`. At that point, you'll run `sudo apt dist-upgrade` to install any remaining updates that didn't get installed with the first command.

In regard to updating the kernel, this process deserves some additional discussion. Some distributions are very risky when it comes to updating the kernel. Arch Linux is an example of this, where only one kernel is installed at any one time. Therefore, when that kernel gets updated, you really need to reboot the machine so that it can use it properly (sometimes, various system components may have difficulty in the case where you have a pending reboot after installing a new kernel).

Ubuntu, on the other hand, handles kernel upgrades very efficiently. When you update a kernel in Ubuntu, it doesn't replace the kernel your server is currently running on. Instead, it installs the updated kernel alongside your existing one.

In fact, these kernels will continue to be stacked and none of them will be removed as new ones are installed. When new versions of the Ubuntu kernel are installed, the GRUB boot loader will be updated automatically to boot the new kernel the next time you perform a reboot. Until you do, you can continue to run on your current kernel for as long as you need to, and you shouldn't notice any difference. The only real difference is the fact that you're not taking advantage of the additional security patches of the new kernel until you reboot, which you can do during your next maintenance window. The reason this method of updating is great is that if you run into a problem where the new kernel doesn't boot or has some sort of issue, you'll have a chance to press *Esc* at the beginning of the boot process, where you'll be able to choose the **Advanced options for Ubuntu** option, which will bring you to a list of all of your installed kernels. Using this list, you can choose between your previous (known, working) kernels and continue to use your server as you were before you updated the kernel. This is a valuable safety net!

After you update the packages on your server, you may want to restart services in order to take advantage of the new security updates. In the case of kernels, you would need to reboot your entire server in order to take advantage of kernel updates, but other updates don't require a reboot. Instead, if you restart the associated service, you'll generally be fine (if the update itself didn't already trigger a restart of a service). For example, if your DNS service (`bind9`) was updated, you would only need to execute the following to restart the service:

```
sudo systemctl restart bind9
```

In addition to keeping packages up to date, it's also important that you understand how to roll back an updated package in a situation where something went wrong. You can recover from such a situation by simply reinstalling an older version of a package manually. Previously downloaded packages are stored in the following directory:

```
/var/cache/apt/archives
```

There, you should find the actual packages that were downloaded as a part of your update process. In a case where you need to restore an updated package to a previously installed version, you can manually install a package with the `dpkg` command. Generally, the syntax will be similar to the following:

```
sudo dpkg -i /path/to/package.deb
```

To be more precise, you would use a command such as the following to reinstall a previously downloaded package, using an older Linux kernel as an example:

```
sudo dpkg -i /var/cache/apt/archives/linux-image-5.4.0-42-
generic_5.4.0-42.46_amd64.deb
```

However, with the `dpkg` command, dependencies aren't handled automatically, so if you are missing a package that your target package requires as a dependency, the package will still be installed, but you'll have unresolved dependencies you'll need to fix. You can try to resolve this situation with `apt`:

```
sudo apt -f install
```

The `apt -f install` command will attempt to fix your installed packages, looking for packages that are missing (but are required by an installed package), and will offer to install the missing dependencies for you. In the case where it cannot find a missing dependency, it will offer to remove the package that requires the missing packages if the situation cannot be worked out any other way.

Well, there you have it. At this point, you should be well on your way to not only installing packages but keeping them updated as well. There's also a feature in Ubuntu that you can utilize to take advantage of the concept of **live patching**, which you can use to patch your server's kernel automatically. That's what we'll cover in the next section.

## Automatically installing patches with the Canonical Livepatch service

In the previous section, I mentioned that if your updates include an update to the kernel, you'll need to reboot your server for the new kernel to take effect. While this is generally true, Canonical offers a **Livepatch** service for Ubuntu, which allows it to receive updates and have them applied without rebooting. This is a game changer, as it takes care of keeping your running kernel patched without you having to do anything, not even reboot. This is a massive benefit to security as it gives you the benefits of the latest security patches without the inconvenience of scheduling a restart of your servers right away.

However, the service is not free or included with Ubuntu by default. Even so, you can install the Livepatch service on three of your servers without paying, so it's still something you may want to consider. You're even able to utilize this service on the desktop version of Ubuntu if you'd like. Since you can use this service for free on three servers, I see no reason why you shouldn't benefit from this on your most critical resources.

Even though you generally won't need to reboot your server in order to take advantage of patches with the Livepatch service, there may be some exceptions depending on the nature of the vulnerability. There have been exploits in the past that required complex changes, and even servers subscribed to this service still needed to reboot. This is the exception rather than the rule, though. Most of the time, a reboot is simply not something you'll need to worry about if you're utilizing Livepatch. More often than not, your server will have all patches applied and inserted right into the running kernel, which is an amazing thing.

One important thing to note is that this doesn't stop you from needing to install updates via `apt`. Live patches are inserted right into the kernel, but they're not permanent. You'll still want to install all of your package updates on a regular basis through the regular means. At the very least, live patches will make it so that you won't be in such a hurry to reboot. If an exploit is revealed on Monday but you aren't able to reboot your server until Sunday, it's no big deal.

Since the Livepatch service requires a subscription, you'll need to create an account in order to get started using it. You can get started with this process at `https://auth.livepatch.canonical.com/`.

The process will involve having you create an Ubuntu One account (`https://login.ubuntu.com/`), which is Canonical's centralized login system. You'll enter your email address, choose a password, and then at the end of the process you'll be given a token to use with your Livepatch service, which will be a string of random characters.

Now that you have a token, you can decide on the three servers that are most important to you. On each of those servers, you can run the following commands to get started:

```
sudo snap install canonical-livepatch
sudo canonical-livepatch enable <token>
```

Believe it or not, that's all there is to it. With how amazing the Livepatch service is, you'd think it would be a complicated process to set up. The most time-consuming part is registering for a new account, as it only takes two commands to set this service up on a server. You can check the status of Livepatch with the following command:

```
sudo canonical-livepatch status
```

Depending on the budget of your organization, you may decide that this service is worth paying for, which will allow you to benefit from having it on more than three servers. It's definitely worth considering. You'll need to contact Canonical to inquire about additional support, should you decide to explore that option.

Another useful concept is that of a dashboard, specifically one we can use to monitor the health of our servers from a graphical user interface. Canonical's Landscape service offers us exactly this, and we'll take a look at it next.

## Monitoring Ubuntu servers with Canonical's Landscape service

Another important aspect of security is keeping track of compliance. Servers simply won't let you know by themselves whether they're behind on updates, and without some sort of service performing some sort of monitoring, you really won't know what's going on with your servers unless you check.

To solve this problem, Canonical offers a custom service known as **Landscape**. Landscape allows you to manage your entire fleet of Ubuntu servers from a single page. Landscape will allow you to list any servers that need security updates, automate common tasks, create your own repositories, and more. It presents these features in an attractive user interface. With such a service, it's easy to tell which of your servers need a security update or a reboot in order to apply a patch.

There are two ways you can utilize Landscape. You can host it yourself (referred to by Canonical as an *on-premises* installation) or you can simply subscribe to Canonical's hosted version. The latter is the absolute easiest way to get on board with Landscape, but there is a cost (currently $0.01 USD per hour). If you install it on-premises, you won't have to pay any fees for the first ten servers, but it will be up to you to maintain it and keep it running.

In this section, I'll walk you through hosting this service yourself. However, there are some important considerations to keep in mind. First, Landscape will have some modest resource requirements. Therefore, a free tier EC2 instance or a lower-cost VPS instance will not be up to the task. At the minimum, you'll want 2 GB of RAM and one CPU core. Second, I advise against setting up Landscape on an existing server alongside other hosted resources. While it's certainly possible to share it with other resources, it may conflict with any web server configuration you may have applied. Lastly, Landscape is only supported with *LTS* editions of Ubuntu. If you're running something other than an LTS release of Ubuntu Server, the required packages won't appear in the repositories.

At the time of writing, the server component of Landscape is not available on Ubuntu 20.04. It is expected to be available at some point in the future, but for now, use Ubuntu 18.04 if you would like to utilize Landscape. The client package is available on 20.04, but specifically the server package has yet to be updated.

The setup of Landscape is relatively easy; there are only two commands to run. The following site lists all the commands required to set up the service:

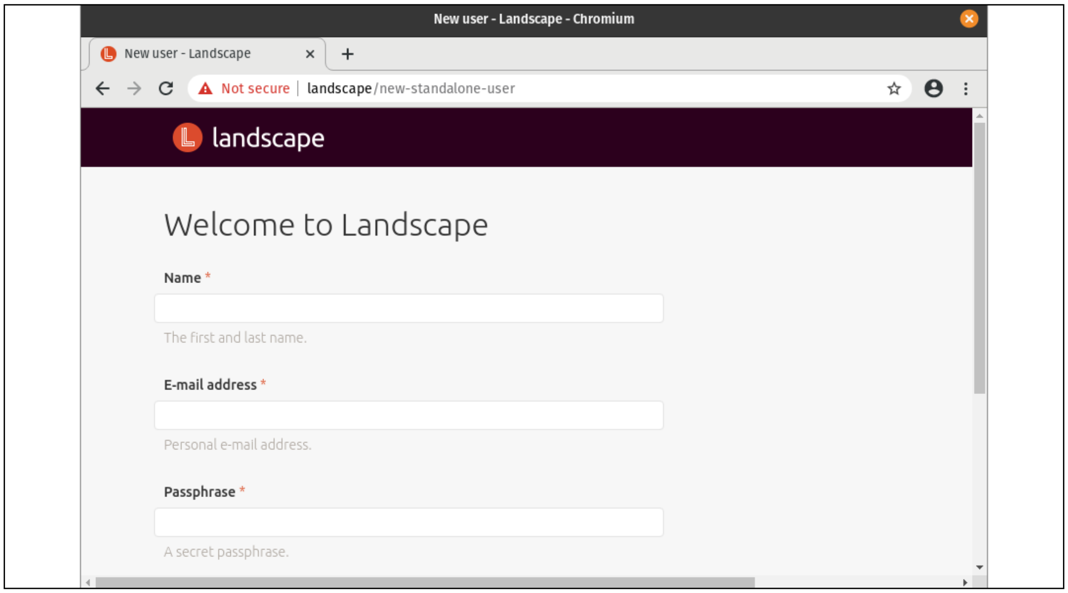
`https://landscape.canonical.com/set-up-on-prem.`

I will list the commands within this section as well, but I also wanted you to have the URL for Canonical's official installation instructions in case any of the commands change for any reason, so if the following commands don't work, check the website. At the time of writing, the following commands will install the Landscape software on your server:

```
sudo add-apt-repository --update ppa:landscape/19.10
sudo apt install landscape-server-quickstart
```

> This process will take a decent chunk of time and install a great deal of dependencies.

Once the process completes, you should be able to access Landscape by simply typing the IP address of your server in a browser window. Ignore the warning regarding SSL, since we haven't actually generated any certificates. You should see a page welcoming you to Landscape, which gives you some fields to fill out:


New user - Landscape - Chromium
New user - Landscape
Not secure
landscape/new-standalone-user
landscape

Welcome to Landscape

Name *

The first and last name.

E-mail address *

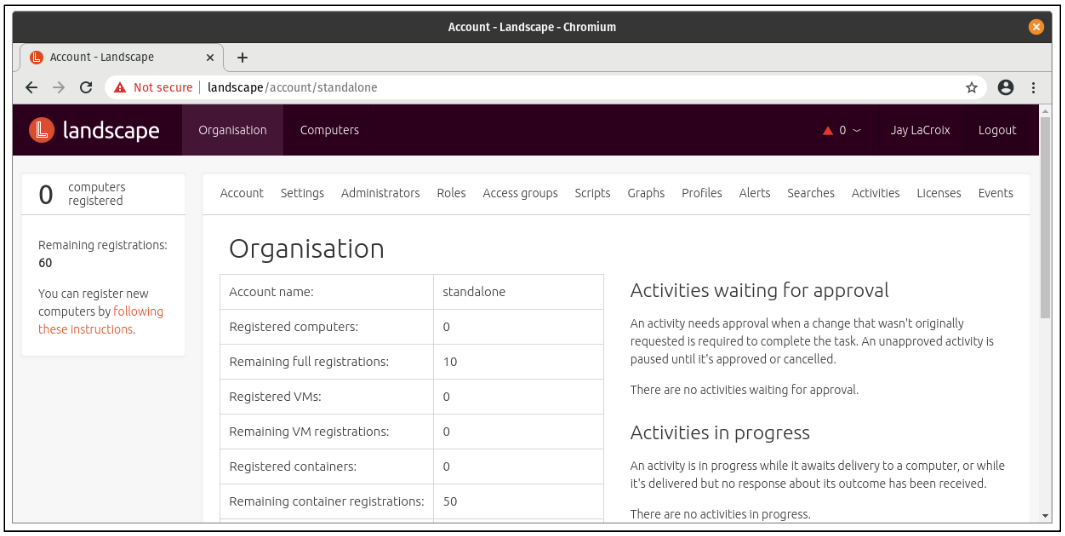
Personal e-mail address.

Passphrase *

A secret passphrase.

Figure 21.3: Setting up Landscape

After filling out the information and continuing on, you should see the main interface for Landscape, but we haven't actually added a system to this service yet, so at this point it's not very useful:


Account - Landscape - Chromium

Account - Landscape

Not secure | landscape/account/standalone

0 computers registered

Remaining registrations: 60

You can register new computers by following these instructions.

Organisation

| | |
|---|---|
| Account name: | standalone |
| Registered computers: | 0 |
| Remaining full registrations: | 10 |
| Registered VMs: | 0 |
| Remaining VM registrations: | 0 |
| Registered containers: | 0 |
| Remaining container registrations: | 50 |

Activities waiting for approval

An activity needs approval when a change that wasn't originally requested is required to complete the task. An unapproved activity is paused until it's approved or cancelled.

There are no activities waiting for approval.

Activities in progress

An activity is in progress while it awaits delivery to a computer, or while it's delivered but no response about its outcome has been received.

There are no activities in progress.

Figure 21.4: The main interface page for Landscape

Adding a system to the Landscape service is quite easy, and the instructions are actually on your Landscape instance itself. To view the instructions, click on **Computers** at the top of the screen, then click on the instructions in the middle of the screen. Or, simply access this URL:

`https://<IP_Address>/account/standalone/how-to-register`

The commands the instructions page will give you will look similar to the following ones:

```
sudo apt update
sudo apt install landscape-client
sudo landscape-config --computer-title "My Server" --account-name
standalone --url https://<IP_ADDRESS>/message-system --ping-url
http://<IP_Address>/ping
```

I don't recommend copying and pasting those exact commands; I just included them as a sample reference. Your Landscape server will instruct you on what to do, so I recommend you follow that. Also, while the Landscape server is only available for 18.04 at the time of this writing, the client is available on Ubuntu 20.04 so you will be able to add your newer servers as clients.

The last of the three commands will ask you a series of questions, and if you're in a hurry, you can accept the defaults for each. I recommend you take your time and read through them though, as some of the optional features may be useful to you. For example, the ability to execute scripts on servers is disabled by default, but you'll be asked whether you want to enable this during the process. You may want to benefit from that feature, but it's not required.

Depending on whether or not you add an actual TLS certificate to your Landscape server, the client installation may fail with SSL errors. We discussed TLS in *Chapter 14, Serving Web Content,* if you need a refresher on that. If this happens, you can work around it by accessing the Landscape server via SSH and using `scp` to copy its certificate file to the server that you're attempting to add as a client:

```
scp /etc/ssl/certs/landscape_server_ca.crt <IP_ADDRESS_OF_CLIENT>:
```

Then, access the client server via OpenSSH, and then change the ownership of the certificate file and move it to the `/etc/landscape` directory:

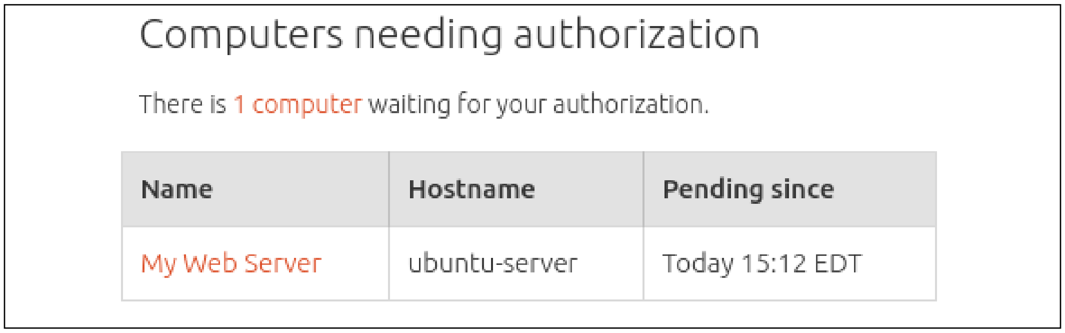
```
sudo chown root:root landscape_server_ca.crt
sudo mv landscape_server_ca.crt /etc/landscape/
```

Finally, edit the `/etc/landscape/client.conf` file and add the following line to the end:

```
ssl_public_key = /etc/landscape/landscape_server_ca.crt
```

If you attempt the registration again, you should have better results.

After you've successfully set up the client on your server, the main page should show an alert that a computer is waiting for approval. It will look something like this:



Computers needing authorization

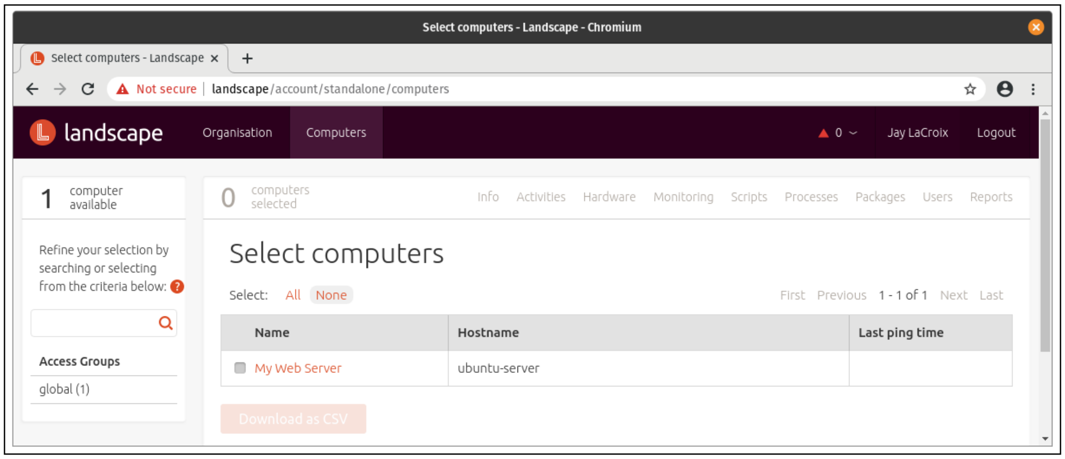
There is 1 computer waiting for your authorization.

| Name | Hostname | Pending since |
|---|---|---|
| My Web Server | ubuntu-server | Today 15:12 EDT |

Figure 21.5: A computer waiting for acceptance in Landscape

To accept the server, click on the server's name (which is shown under the **Name** section), which will bring you to another page that will allow you to configure details, such as its name and tags. Click the **Accept** button on that page to finish the process. The server will now be listed on the Landscape homepage, and you'll be able to interact with it within the interface.

To get started with managing the servers you've added, click on the **Computers** tab at the very top of the page. A list of all the servers associated with your Landscape server will appear there:



| Name | Hostname | Last ping time |
|---|---|---|
| My Web Server | ubuntu-server | |

Figure 21.6: Landscape's server list

Next, select a server by clicking on the text underneath the **Name** column that corresponds to the server you'd like to manage. This will bring you to a page specific to that server, showing you various details about it as well as giving you a menu of actions you can perform against it. The menu will look like the following:

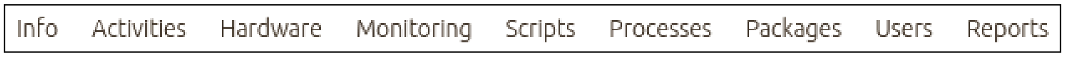


Figure 21.7: The Landscape computer menu

Here's a list of the common items in this menu and a short description of what each one does:

- **Info**: Shows basic information about the server, such as what model of processor it has, RAM, and other system information.
- **Activities**: This section shows the status of any commands you've given the server to perform through the interface, such as installing a package.
- **Hardware**: Provides more specific details about hardware devices installed on the server.

- **Monitoring**: This section allows you to view information regarding resource utilization.
- **Scripts**: If enabled, you'll be able to paste a script here in a format such as Bash and have it run against the server.
- **Processes**: Return a list of processes running on the server. You can end and even kill processes here as well.
- **Packages**: Here, you can search for a package, and even install it on the server without having to enter a single shell command.
- **Users**: Here, you can create, delete, and modify users on the system.
- **Reports**: In this section, you can report on specific information regarding your server. Most notably for our purposes in this chapter, you can report on security update compliance.

As I'm sure you can see, you can do some really neat things within Landscape. We only scratched the surface in this section. This service provides you with a central interface through which you can manage your servers. You can manage users, run scripts, install packages, and more. I'll leave it up to you to spend some time thoroughly exploring Landscape to experiment with all of the things you can do with it. In addition, I'll include a link to the official documentation pages at the end of this chapter if you would like to explore this service even more.

At this point, we should switch gears and discuss some things we can do to better secure OpenSSH. I've mentioned a few times throughout this chapter that OpenSSH is a common target for outside threat actors, so in the next section, it's time to take a closer look at this.

## Securing OpenSSH

OpenSSH is a very useful utility; it allows us to configure our servers from a remote location as if we were sitting in front of the console. In the case of cloud resources, it's typically the only way to access our servers. Considering the nature of OpenSSH itself (remote administration), it's a very tempting target for miscreants who are looking to cause trouble. If we simply leave OpenSSH unsecured, this useful utility may be our worst nightmare.

Thankfully, configuring OpenSSH itself is very easy. However, the large number of configuration options may be intimidating to someone who doesn't have much experience tuning it. While it's a good idea to peruse the documentation for OpenSSH, in this section, we'll take a look at the common configuration options you'll want to focus your attention on first.

The configuration file for OpenSSH itself is located at `/etc/ssh/sshd_config`, and we touched on it in *Chapter 10, Connecting to Networks*. This is the file we're going to focus on in this section, as the configuration options I'm going to give you are to be placed in that file.

With each of the tweaks in this section, make sure you first look through the file in order to see whether the setting is already there and change it accordingly. If the setting is not present in the file, add it. After you make your changes, it's important to restart the OpenSSH daemon:

```
sudo systemctl restart ssh
```

Go ahead and open this file in your editor, and we'll go through some tweaks.

One really easy tweak is to change the port number that OpenSSH listens on, which defaults to port `22`. Since this is the first port that hackers will attempt, it makes sense to change it, and it's a very easy change to make. However, I don't want you to think that just because you change the port for OpenSSH, it's magically hidden and cannot be detected. A persistent hacker will still be able to find the port by running a port scan against your server. However, with the change being so easy to tweak, why not do it? To change it, simply look for the port number in the `/etc/ssh/sshd_config` file and change it from its default of `22`:

```
Port 65332
```

The only downsides I can think of in regards to changing the SSH port are that you'll have to remember to specify the port number when using SSH, and you'll have to communicate the change to anyone that uses the server. To specify the port, we use the `-p` option with the `ssh` command:

```
ssh -p 65332 myhost
```

If you're using `scp`, you'll need to use an uppercase `P` instead:

```
scp -P 65332 myfile myserver:/path/to/dir
```

Even though changing the port number won't make your server bulletproof, we shouldn't underestimate the value of doing so. In a hypothetical example where an attacker is scanning servers on the internet for an open port `22`, they'll probably skip your server and move on to the next. Only determined attackers that specifically want to break into your server will scan other ports looking for it. This also keeps your log file clean; you'll see intrusion attempts only from miscreants doing aggressive port scans, rather than random bots looking for open ports.

If your server is internet-facing, this will result in far fewer entries in the logs! OpenSSH logs connection attempts in the authorization log, located at `/var/log/auth.log`. Feel free to check out that log file to see what typical logging looks like.

Another change that's worth mentioning is which protocol OpenSSH listens for. Most versions of OpenSSH available in repositories today default to Protocol 2. This is what you want. Protocol 2 is much more secure than Protocol 1. You should never allow Protocol 1 in production under any circumstances. Chances are you're probably already using the default of Protocol 2 on your server, unless you changed it for some reason. I mention it here just in case you have older servers still in production that are defaulting to the older protocol. Nowadays, OpenSSH is always on Protocol 2 in any modern release of a Linux distribution. If you do have an older server that's still using Protocol 1, you can adjust that by finding the following line in the `/etc/ssh/sshd_config` file:

```
Protocol 1
```

Switching OpenSSH to use Protocol 2 is as simple as changing the `1` on that line to `2`, and then restarting the OpenSSH server:

```
sudo systemctl restart ssh
```

Next, I'll give you two tweaks for the price of one. There are two settings that deal with which users and groups are allowed to log in via SSH: `AllowUsers` and `AllowGroups`, respectively. By default, every user you create is allowed to log in to your server via SSH. With regards to `root`, that's actually not allowed by default (more on that later). But each user you create is allowed in. However, only users that must have access should be allowed in. There are two ways to accomplish this.

One option is to use `AllowUsers`. With the `AllowUsers` option, you can specifically set which users can log in to your server. With `AllowUsers` present (which is not found in the `config` file by default), your server will not allow anyone to use SSH that you don't specifically call out with that option. You can separate each user with a space:

```
AllowUsers larry moe curly
```

Personally, I find `AllowGroups` easier to manage. It works pretty much the same as `AllowUsers`, but with groups. If present, it will restrict OpenSSH connections to users who are a member of this group. To use it, you'll first create the group in question (you can name it whatever makes sense to you):

```
sudo groupadd sshusers
```

Then, you'll make one or more users a member of that group:

```
sudo usermod -aG sshusers myuser
```

Once you have added the group and made a user or two a member of that group, add the following to your `/etc/ssh/sshd_config` file, replacing the sample groups with yours:

```
AllowGroups admins sshusers gremlins
```

It's fine to use only one group. Just make sure you add yourself to the group before you log out, otherwise you'll lock yourself out. I recommend you use only one or the other between `AllowUsers` and `AllowGroups`. I think that it's much easier to use `AllowGroups`, since you'll never need to touch the `sshd_config` file again; you'll simply add or remove user accounts to and from the group to control access. Just so you're aware, `AllowUsers` overrides `AllowGroups`.

Another important option is `PermitRootLogin`, which controls whether or not the `root` user account is able to make SSH connections. This should always be set to `no`. By default, this is usually set to `prohibit-password`, which means key authentication is allowed for `root` while passwords for `root` aren't accepted. I don't see any reason for this either. In my opinion, you should turn this off. Having `root` being able to log in to your server over a network connection is never a good idea. This is always the first user account attackers will try to use:

```
PermitRootLogin no
```

There is one exception to the *no-root* rule with SSH. Some providers of cloud servers, such as Linode, may have you log in as `root` by default. This isn't really typical, but some providers are set up that way. In such a case, I recommend creating a regular user with `sudo` access, and then disallowing `root` login.

My next suggestion is by no means easy to set up, but it's worth it. By default, OpenSSH allows users to authenticate via passwords. This is one of the first things I disable on all my servers. Allowing users to enter passwords to establish a connection means that attackers will also be able to brute-force your server. If passwords aren't allowed, then they can't do that. What's tricky is that before you can disable password authentication for SSH, you'll first need to configure and test an alternate means of authenticating, which will usually be public key authentication. This is something we've gone over, in *Chapter 10, Connecting to Networks*. Basically, you can generate an SSH key pair on your local workstation, and then add that key to the `authorized_keys` file on the server, which will allow you in without a password. Again, refer to *Chapter 10, Connecting to Networks*, if you haven't played around with this yet.

If you disable password authentication for OpenSSH, then public key authentication will be the only way in. If someone tries to connect to your server and they don't have the appropriate key, the server will deny their access immediately. If password authentication is enabled and you have a key relationship, then the server will ask the user for their password if their key isn't installed. In my view, after you set up access via public key cryptography, you should disable password authentication (just make sure you test it first):

```
PasswordAuthentication no
```

There you are, those are my most recommended tweaks for securing OpenSSH. There's certainly more where that came from, but those are the settings you'll benefit from the most. In the next section, we'll add an additional layer, in the form of Fail2ban. With Fail2ban protecting OpenSSH and coupled with the tweaks I mentioned in this section, attackers will have a tough time trying to break into your server. For your convenience, here are all the OpenSSH configuration options I've covered in this section:

```
Port 65332
Protocol 2
AllowUsers larry moe curly
AllowGroups admins sshusers gremlins
PermitRootLogin no
PasswordAuthentication no
```

With OpenSSH better secured, we should be a bit more confident now when it comes to the security of our server. However, each tweak or improvement we make to improve security only helps us so much. The more protections we implement, the better. In the next section, we'll explore Fail2ban, which can greatly increase the security of our server.

## Installing and configuring Fail2ban

Fail2ban, how I love thee! **Fail2ban** is one of those tools that once I learned how valuable it is, I wondered how I ever lived so long without it. Fail2ban is able to keep an eye on your log files, looking for authentication failures. You can set the number of failures that are allowed from any given IP address, and if there are more than the allowed number of failures, Fail2ban will block that individual's IP address. It's highly configurable and can enhance the security of your server.

Installing and configuring Fail2ban is relatively straightforward. First, install its package:

```
sudo apt install fail2ban
```

After installation, the `fail2ban` daemon will start up and be configured to automatically start at boot time. Configuring `fail2ban` is simply a matter of creating a configuration file. But this is one of the more interesting aspects of Fail2ban: you shouldn't use its default `config` file. The default file is `/etc/fail2ban/jail.conf`. The problem with this file is that it can be overwritten when you install security updates, if those security updates ever include Fail2ban itself. To remedy this, Fail2ban also reads the `/etc/fail2ban/jail.local` file, if it exists. It will never replace that file, and the presence of a `jail.local` file will supersede the `jail.conf` file. The simplest way to get started is to make a copy of `jail.conf` and save it as `jail.local`:

```
sudo cp /etc/fail2ban/jail.conf /etc/fail2ban/jail.local
```

Next, I'll go over some of the very important settings you should configure, so open up the `/etc/fail2ban/jail.local` file you just copied in a text editor. The first configuration item to change is located on or around line 92 and is commented out:

```
#ignoreip = 127.0.0.1/8 ::1
```

First of all, uncomment it. Then, you should add additional networks that you don't want to be blocked by Fail2ban. Basically, this will help prevent you from getting locked out in a situation where you accidentally trigger Fail2ban. Fail2ban is relentless; it will block any service that meets its block criteria, and it won't think twice about it. This includes blocking you. To rectify this, add your company's network here, as well as any other IP address you never want to be blocked. Make sure to leave the `localhost` IP intact:

```
Ignoreip = 127.0.0.1/8 ::1 192.168.1.0/24 192.168.1.245/24
```

In that example, I added the `192.168.1.0/24` network, as well as a single IP address of `192.168.1.245/24`. Add your networks to this line to *ensure you don't lock yourself out*.

Next, line 101 includes the `bantime` option. This option pertains to how many seconds a host is banned when Fail2ban blocks it. This option defaults to `10m`, or 10 minutes:

```
bantime  = 10m
```

Change this number to whatever you find reasonable, or just leave it as its default, which will also be fine. If a host gets banned, it will be banned for this specific number of minutes, then it will eventually be allowed again.

Continuing, we have the `maxretry` setting:

```
maxretry = 5
```

This is specifically the number of failures that need to occur before Fail2ban takes action. If a service it's watching reaches the number set here, game over! The IP will be blocked for the number of minutes included in the `bantime` option. You can change this if you want to, if you don't find `5` failures to be reasonable. The highest I would set it to is `7`, for those users on your network who insist they're typing the correct password and they type the same (wrong) thing over and over. Hopefully, they'll realize their error before their seventh attempt and won't need to call the helpdesk.

Skipping ahead all the way down to line 272 or thereabouts, we have the `Jails` section. From here, the `config` file will list several jails you can configure, which is basically another word for something Fail2ban will pay attention to. The first is `[sshd]`, which configures its protection of the OpenSSH daemon. Look for this option underneath `[sshd]`:

```
port    = ssh
```

`port` being equal to `ssh` basically means that it's defaulting to port `22`. If you've changed your SSH port, change this to reflect whatever that port is. There are two such occurrences, one under `[sshd]` and another underneath `[sshd-ddos]`:

```
port    = 65332
```

Before we go too much further, I want to underscore the fact that we should test whether Fail2ban is working after each configuration change we make. To do this, restart Fail2ban and then check its status:

```
sudo systemctl restart fail2ban
sudo systemctl status -l fail2ban
```

The status should always be `active (running)`. If it's anything else (such as `failed`), that means that Fail2ban doesn't like something in your configuration. Usually, that means that Fail2ban's status will reflect that it exited. So, as we go, make sure to restart Fail2ban after each change and make sure it's not complaining about something. The `status` command will show lines from Fail2ban's log file for your convenience.

Another useful command to run after restarting Fail2ban is the following:

```
sudo fail2ban-client status
```

The output from that command will show all the jails that you have enabled. If you enable a new jail in the `config` file, you should see it listed within the output of that command.

So, how do you enable a jail? By default, all jails are disabled, except for the one for OpenSSH. To enable a jail, place the following within its `config` block in the `/etc/fail2ban/jail.local` file:

```
enabled = true
```

If you want to enable the `apache-auth` jail, find its section, and place `enabled = true` right underneath its stanza. For example, `apache-auth` will look like the following after you add the `enabled` line:

```
[apache-auth]
enabled = true
port     = http,https
logpath  = %(apache_error_log)
```

In that example, the `enabled = true` portion wasn't present in the default file. I added it. Now that I've enabled a new jail, we should restart `fail2ban`:

```
sudo systemctl restart fail2ban
```

Next, check its status to make sure it didn't explode on startup:

```
sudo systemctl status -l fail2ban
```

Assuming all went well, we should see the new jail listed in the output of the following command:

```
sudo fail2ban-client status
```

On my test server, the output became the following once I enabled `apache-auth`:

```
Status
|- Number of jail: 2

  `- Jail list:    apache-auth, sshd
```

If you enable a jail for a service you don't have installed, Fail2ban may fail to start up. In my example, I actually did have `apache2` installed on that server before I enabled its jail. If I hadn't, Fail2ban would likely have exited, complaining that it wasn't able to find log files for Apache. This is the reason why I recommend that you test Fail2ban after enabling any jail. If Fail2ban decides it doesn't like something, or something it's looking for isn't present, it may stop. Then, it won't be protecting you at all, which is not good.

The basic order of operations for Fail2ban is to peruse the jail `config` file, looking for any jails you may benefit from. If you have a daemon running on your server, there's a chance that there's a jail for that. If there is, enable it and see whether Fail2ban breaks. If not, you're in good shape. If it does fail to restart properly, inspect the status output and check what it's complaining about.

One thing you may want to do is add the `enabled = true` line to `[sshd]` and `[sshd-ddos]`. Sure, the `[sshd]` jail is already enabled by default, but since it wasn't specifically called out in the `config` file, I don't trust it. So you might as well add an `enabled` line to be safe. There are several jails you may benefit from. If you are using `SSL` with `Apache`, enable `[apache-modsecurity]`. Also, consider enabling `[apache-shellshock]` while you're at it to potentially protect Apache from the Shell Shock vulnerability. If you're running your own mail server and have `Roundcube` running, enable `[roundcube-auth]` and `[postfix]`. There are a lot of default jails at your disposal!

Like all security applications, Fail2ban isn't going to automatically make your server impervious to all attacks, but it is a helpful additional layer you can add to your security regimen. When it comes to the jails for OpenSSH, Fail2ban is worth its weight in gold, and that's really the least you should enable. Go ahead and give Fail2ban a go on your servers—just make sure you also add your own network to the `Ignoreip` list that was covered earlier, in case you accidentally type your own SSH password incorrectly too many times and potentially lock yourself out. Fail2ban doesn't discriminate; it'll block anyone. Once you get it fully configured, I think you'll agree that Fail2ban is a worthy ally for your servers.

Earlier, I mentioned that each service that runs on your computer listening for connections is a potential target. While it's impossible to go over every service you could possibly run on your server and how to secure it, we will want to consider securing our database server (if we have one) since organizations typically store valuable data there. We'll learn some methods we can utilize to better secure MariaDB next.

## MariaDB best practices for secure database servers

MariaDB, as well as MySQL, is a very useful resource to have at your disposal. However, it can also be used against you if configured improperly. Thankfully, it's not too hard to secure, but there are several points of consideration to make regarding your database server when developing your security design.

The first point is probably obvious to most of you, and I have mentioned it before, but I'll mention it just in case. Your database server should not be reachable from the internet. I do understand that there are some edge cases when developing a network, and certain applications may require access to a MySQL database over the internet. However, if your database server is accessible over the internet, miscreants will try their best to attack it and gain entry. If there's any vulnerability in your version of MariaDB or MySQL, they'll most likely be able to hack into it.

In most organizations, a great way to implement a database server is to make it accessible by only internal servers. This means that while your web server would obviously be accessible from the internet, its backend database should exist on a different server on your internal network and accept communications only from the web server. If your database server is a VPS or cloud instance, it should especially be configured to only accept communications from your web server, as VPS machines are accessible via the internet by default. Therefore, it's still possible for your database server to be breached if your web server is also breached, but it would be less likely to be compromised if it resides on a separate and restricted server.

> Some VPS providers, such as DigitalOcean and Linode, feature local networking, which you can leverage for your database server instead of allowing it to be accessible over the internet. If your VPS provider features local networking, you should definitely utilize it and deny traffic from outside the local network.

With regards to limiting which servers are able to access a database server, there are a few tweaks we can use to accomplish this. First, we can leverage the `/etc/hosts.allow` and `/etc/hosts.deny` files. With the `/etc/hosts.deny` file, we can stop traffic from certain networks or from specific services. With `/etc/hosts.allow`, we allow the traffic. This works because IP addresses included in `/etc/hosts.allow` override `/etc/hosts.deny`. So basically, if you deny everything in `/etc/hosts.deny` and allow a resource or two in `/etc/hosts.allow`, you're saying, deny everything, except resources I explicitly allow from the `/etc/hosts.deny` file.

To make this change, we'll want to edit the /etc/hosts.allow file first. By default, this file has no configuration other than some helpful comments. Within the file, we can include a list of resources we'd like to be able to access our server, no matter what. Make sure that you include your web server here, and also make sure that you immediately add the IP address you'll be using to SSH into the machine, otherwise you'll lock yourself out once we edit the /etc/hosts.deny file. Here are some example hosts.allow entries, with a description of what each example rule does.

The first example rule allows a machine with an IP address of 192.168.1.50 to access the server:

```
ALL: 192.168.1.50
```

This rule allows any machine within the 192.168.1.0/24 network to access the server:

```
ALL: 192.168.1.0/255.255.255.0
```

In this rule, we have an incomplete IP address. This acts as a wildcard, which means that any IP address beginning with 192.168.1 is allowed:

```
ALL: 192.168.1.
```

This rule allows everything. You definitely don't want to do this:

```
ALL: ALL
```

We can also allow specific daemons. Here, I'm allowing OpenSSH traffic originating from any IP address beginning with 192.168.1:

```
ssh: 192.168.1.
```

On your end, if you wish to utilize this security approach, add the resources on the database server you'll be comfortable accepting communications from. Make sure you at least add the IP address of another server with access to OpenSSH, so you'll have a way to manage the machine. You can also add all your internal IP addresses with a rule similar to the previous examples. Once you have this set up, we can edit the /etc/hosts.deny file.

The /etc/hosts.deny file utilizes the same syntax as /etc/hosts.allow. To finish this little exercise, we can block any traffic not included in the /etc/hosts.allow file with the following rule:

```
ALL: ALL
```

The /etc/hosts.allow and /etc/hosts.deny files don't represent a complete layer of security but are a great first step in securing a database server, especially one that might contain sensitive user or financial information. They're by no means specific to MySQL either, but I mention them here because databases very often contain data that, if leaked, could potentially wreak havoc on your organization and even put someone out of business. A database server should only ever be accessible by the application that needs to utilize it.

Another point of consideration is user security. We walked through creating database users in *Chapter 13, Managing Databases*. In that chapter, we walked through the MySQL commands for creating a user as well as GRANT, performing both in one single command. This is the example I used:

```
GRANT SELECT ON mysampledb.* TO 'appuser'@'localhost' IDENTIFIED BY
'password';
```

What's important here is that we're allowing access to the mysampledb database by a user named appuser. If you look closer at the command, we're also specifying that this connection is allowed only if it's coming in from localhost. If we tried to access this database remotely, it wouldn't be allowed. This is a great default. But you'll also, at some point, need to access the database from a different server. Perhaps your web server and database server are separate machines, which is a common enterprise. You could do this:

```
GRANT SELECT ON mysampledb.* TO 'appuser'@'%' IDENTIFIED BY 'password';
```

However, in my opinion, this is a very bad practice. The % character in a MySQL GRANT command is a wildcard, similar to * with other commands. Here, we're basically telling our MariaDB or MySQL instance to accept connections from this user, from any network. There is almost never a good reason to do this. I've heard some administrators use the argument that they don't allow external traffic from their company firewall, so allowing MySQL traffic from any machine shouldn't be a problem. However, that logic breaks down when you consider that if an attacker does gain access to any machine in your network, they can immediately target your database server. If an internal employee gets angry at management and wants to destroy the database, they'll be able to access it from their workstation. If an employee's workstation becomes affected by malware that targets database servers, it may find your database server and try to brute-force it. I could go on and on with examples of why allowing access to your database server from any machine is a bad idea. Just don't do it!

If we want to give access to a specific IP address, we can do so with the following instead:

```
GRANT SELECT ON mysampledb.* TO 'appuser'@'192.168.1.50' IDENTIFIED BY
'password';
```

With the previous example, only a server or workstation with an IP address of `192.168.1.50` is allowed to use the `appuser` account to obtain access to the database. That's much better. You can, of course, allow an entire subnet as well:

```
GRANT SELECT ON mysampledb.* TO 'appuser'@'192.168.1.% IDENTIFIED BY
'password';
```

Here, any IP address beginning with `192.168.1` is allowed. Honestly, I really don't like allowing an entire subnet. But depending on your network design, you may have a dozen or so machines that need access. Hopefully, the subnet you allow is not the same subnet your users' workstations use!

Finally, another point of consideration is security patches for your database server software. I know I talk about updates quite a bit, but as I've mentioned, these updates exist for a reason. Developers don't release patches for enterprise software simply because they're bored, these updates often patch real problems that real people are taking advantage of right now as you read this. Install updates regularly. I understand that updates on server applications can scare some people, as an update always comes with the risk that it may disrupt business. But as an administrator, it's up to you to create a roll-out plan for security patches, and ensure they're installed in a timely fashion. Sure, it's tough and often has to be done after-hours. But the last thing I want to do is read about yet another company where the contents of their database server were leaked and posted freely online. A good security design includes regular patching.

Now that our database server is more secure, there's another topic worth diving into, and that is the subject of implementing a firewall. There are several different firewall solutions out there, but UFW is a great choice. It's easy to set up, and quite effective. In the next section, I'll go over how to implement it.

## Setting up a firewall

Firewalls are a very important aspect to include in your network and security design. Firewalls are extremely easy to implement, but sometimes hard to implement well. The problem with firewalls is that they can sometimes offer a false sense of security to those who aren't familiar with the best ways to manage them. Sure, they're good to have, but simply having a firewall isn't enough by itself.

The false sense of security comes when someone thinks that they're protected just because a firewall is installed and enabled, but they're also often opening traffic from any network to internal ports. Take into consideration the firewall that was introduced with Windows XP and enabled by default with Windows XP Service Pack 2. Yes, it was a good step but users simply clicked the "allow" button whenever something wanted access, which defeats the entire purpose of having a firewall. Windows implements this better nowadays, but the false sense of security it created remains. Firewalls are not a "set it and forget it" solution!

Firewalls work by allowing or disallowing access to a network port from other networks. Most good firewalls block outside traffic by default. When a user or administrator enables a service, they open a port for it. Then, that service is allowed in. This is great in theory, but where it breaks down is that administrators will often allow access from everywhere when they open a port. If an administrator does this, they may as well not have a firewall at all. If you need access to a server via OpenSSH, you may open up port `22` (or whatever port OpenSSH is listening on) to allow it through the firewall. But if you simply allow the port, it's open for everyone else as well.

When configured properly, a firewall will enable access to a port only from specific places. For example, rather than allowing port `22` for OpenSSH to your entire network, why not just allow traffic to port `22` from specific IP addresses or subnets? Now we're getting somewhere! In my opinion, allowing all traffic through a port is usually a bad idea, though some services actually do need this (such as web traffic to your web server). If you can help it, only allow traffic from specific networks when you open a port. This is where the use case for a firewall really shines.

In Ubuntu Server, the **Uncomplicated Firewall** (**UFW**) is a really useful tool for configuring your firewall. As the name suggests, it makes firewall management a breeze. To get started, install the `ufw` package:

```
sudo apt install ufw
```

By default, the UFW firewall is inactive. This is a good thing, because we wouldn't want to enable a firewall until after we've configured it. The `ufw` package features its own command for checking its status:

```
sudo ufw status
```

Unless you've already configured your firewall, the status will come back as inactive.

With the `ufw` package installed, the first thing we'll want to do is enable traffic via SSH, so we won't get locked out when we do enable the firewall:

```
sudo ufw allow from 192.168.1.156 to any port 22
```

You can probably see from that example how easy UFW's syntax is. With that example, we're allowing the `192.168.1.156` IP address access to port `22` via TCP as well as UDP. In your case, you would change the IP address accordingly, as well as the port number if you're not using the OpenSSH default port. The `any` option refers to any protocol (TCP or UDP).

You can also allow traffic by subnet:

```
sudo ufw allow from 192.168.1.0/24 to any port 22
```

Although I don't recommend this, you can allow all traffic from a specific IP to access anything on your server. Use this with care, if you have to use it at all:

```
sudo ufw allow from 192.168.1.50
```

Now that we've configured our firewall to allow access via OpenSSH, you should also allow any other ports or IP addresses that are required for your server to operate efficiently. If your server is a web server, for example, you'll want to allow traffic from ports `80` and `443`. This is one of those few exceptions where you'll want to allow traffic from any network, assuming your web server serves an external page on the internet:

```
sudo ufw allow 80
sudo ufw allow 443
```

There are various other use patterns for the `ufw` command; refer to the main page (`http://manpages.ubuntu.com/manpages/focal/man8/ufw.8.html`) for more. In a nutshell, these examples should enable you to allow traffic through specific ports, as well as via specific networks and IP addresses. Once you've finished configuring the firewall, we can enable it:

```
sudo ufw enable
Firewall is active and enabled on system startup
```

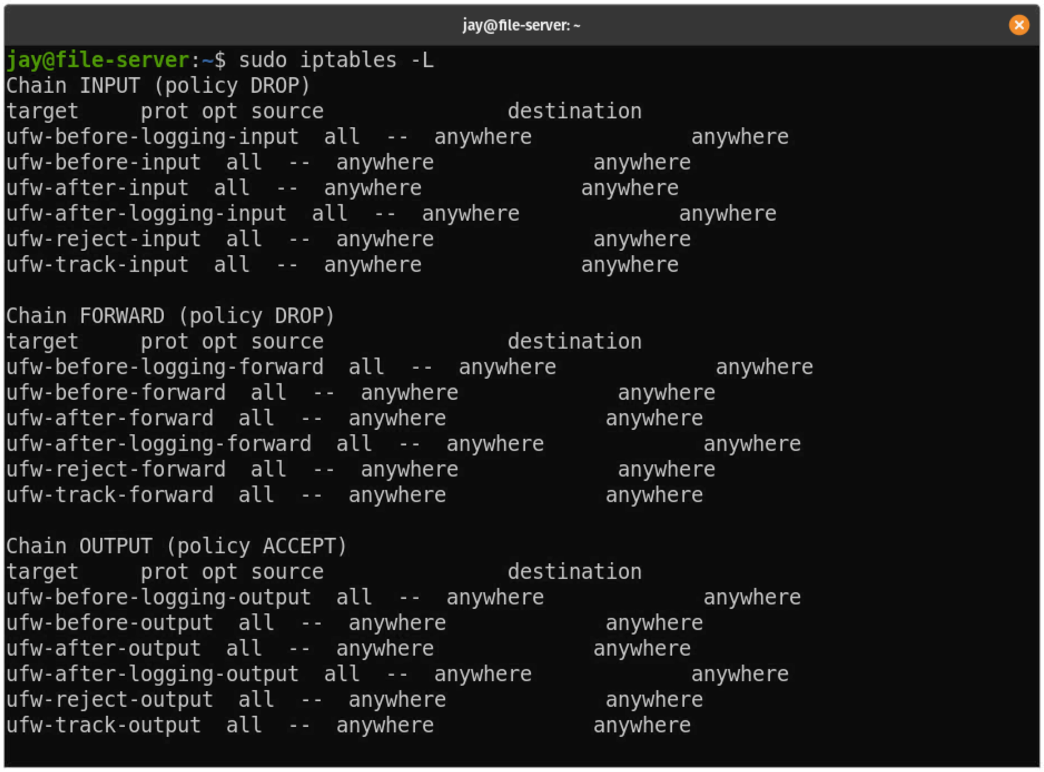
Just as the output suggests, our firewall is active and will start up automatically whenever we reboot the server.

The UFW package is basically an easy-to-use frontend to the `iptables` firewall, and it acts as the default firewall for Ubuntu. The commands we've executed so far in this section trigger the `iptables` command, which is a command that administrators can use to set up a firewall manually. A full walk-through of `iptables` is outside the scope of this chapter, and it's essentially unnecessary, since Ubuntu features UFW as its preferred firewall administration tool and it's the tool you should use while administering a firewall on your Ubuntu server.

If you're curious, you can see what your current set of `iptables` firewall rules look like with the following command:

```
sudo iptables -L
```

The output will show all of the firewall rules that are active on the system, including those created via UFW:



```
jay@file-server:~$ sudo iptables -L
Chain INPUT (policy DROP)
target     prot opt source               destination
ufw-before-logging-input  all  --  anywhere             anywhere
ufw-before-input  all  --  anywhere             anywhere
ufw-after-input  all  --  anywhere             anywhere
ufw-after-logging-input  all  --  anywhere             anywhere
ufw-reject-input  all  --  anywhere             anywhere
ufw-track-input  all  --  anywhere             anywhere

Chain FORWARD (policy DROP)
target     prot opt source               destination
ufw-before-logging-forward  all  --  anywhere             anywhere
ufw-before-forward  all  --  anywhere             anywhere
ufw-after-forward  all  --  anywhere             anywhere
ufw-after-logging-forward  all  --  anywhere             anywhere
ufw-reject-forward  all  --  anywhere             anywhere
ufw-track-forward  all  --  anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ufw-before-logging-output  all  --  anywhere             anywhere
ufw-before-output  all  --  anywhere             anywhere
ufw-after-output  all  --  anywhere             anywhere
ufw-after-logging-output  all  --  anywhere             anywhere
ufw-reject-output  all  --  anywhere             anywhere
ufw-track-output  all  --  anywhere             anywhere
```

Figure 21.8: Viewing firewall rules

With a well-planned firewall implementation, you can better secure your Ubuntu Server installation from outside threats. Preferably, each port you open should only be accessible from specific machines, with the exception being servers that are meant to serve data or resources to external networks. Like all security solutions, a firewall won't make your server invincible, but it does represent an additional layer that attackers would have to bypass in order to do harm.

If your company stores sensitive information, it's important to ensure the storage underneath that data is encrypted. Next, we're going to look at **Linux Unified Key Setup** (**LUKS**), which will help us encrypt and decrypt disks.

## Encrypting and decrypting disks with LUKS

An important aspect of security that many people don't even think about is encryption. As I'm sure you know, backups are essential for business continuity. If a server breaks down, or a resource stops functioning, backups will be your saving grace. But what happens if your backup medium gets stolen or somehow falls into the wrong hands? If your backup is not encrypted, then anyone will be able to view its contents. Some data isn't sensitive, so encryption isn't always required. But anything that contains personally identifiable information, company secrets, or anything else that would cause any kind of hardship if leaked, should be encrypted. In this section, I'll walk you through setting up **LUKS** encryption on an external backup drive.

Before we get into that though, I want to quickly mention the importance of full-disk encryption for your distribution as well. Although this section is going to go over how to encrypt external disks, it's possible to encrypt the volume for your entire Linux installation as well. In the case of Ubuntu, full-disk encryption is an option during installation, for both the server and workstation flavors. This is especially important when it comes to mobile devices, such as laptops, which are stolen quite frequently. If a laptop is planned to store confidential data that you cannot afford to have leaked out, you should choose the option during installation to encrypt your entire Ubuntu installation. If you don't, anyone that knows how to boot a Live OS disc and mount a hard drive will be able to view your data. I've seen unencrypted company laptops get stolen before, and it's not a wonderful experience.

Anyway, back to the topic of encrypting external volumes. For the purpose of encrypting disks, we'll need to install the `cryptsetup` package:

```
sudo apt install cryptsetup
```

The `cryptsetup` utility allows us to encrypt and unencrypt disks. To continue, you'll need an external disk you can safely format, as encrypting the disk will remove any data stored on it. This can be an external hard disk or a flash drive. Both can be treated the exact same way. In addition, you can use this same process to encrypt a secondary internal hard disk attached to your virtual machine or server. I'm assuming that you don't care about the contents saved on the drive, because the process of setting up encryption will wipe it.

If you're using an external disk, use the `fdisk -l` command as `root` or the `lsblk` command to view a list of hard disks attached to your computer or server before you insert it. After you insert your external disk or flash drive, run the command again to determine the device designation for your removable media.

In my examples, I used `/dev/sdb`, but you should use whatever designation your device was given. This is important, because you don't want to wipe out your `root` partition or an existing data partition!

First, we'll need to use `cryptsetup` to format our disk:

```
sudo cryptsetup luksFormat /dev/sdb
```

You'll receive the following warning:

```
WARNING!
========
This will overwrite data on /dev/sdb irrevocably.
Are you sure? (Type uppercase yes):
```

Type `YES` and press *Enter* to continue. Next, you'll be asked for the passphrase. This passphrase will be required in order to unlock the drive. Make sure you use a good, randomly generated password and that you store it somewhere safe. If you lose it, you will not be able to unlock the drive. You'll be asked to confirm the passphrase.

Once the command completes, we can format our encrypted disk. At this point, it has no filesystem, so we'll need to create one. First, open the disk with the following command:

```
sudo cryptsetup luksOpen /dev/sdb backup_drive
```

The `backup_drive` name can be anything you want; it's just an arbitrary name you can refer to the disk as. At this point, the disk will be attached to `/dev/mapper/disk_name`, where `disk_name` is whatever you called your disk in the previous command (in my case, `backup_drive`). Next, we can format the disk. The following command will create an ext4 filesystem on the encrypted disk:

```
sudo mkfs.ext4 -L "backup_drive" /dev/mapper/backup_drive
```

The `-L` option allows us to add a label to the drive, so feel free to change that label to whatever you prefer to name the drive.

With the formatting out of the way, we can now mount the disk:

```
sudo mount /dev/mapper/backup_drive /media/backup_drive
```

The `mount` command will mount the encrypted disk located at `/dev/mapper/backup_drive` and attach it to a mount point, such as `/media/backup_drive` in my example. The target mount directory must already exist. With the disk mounted, you can now save data onto the device as you would any other volume. When finished, you can unmount the device with the following commands:

```
sudo umount /media/backup_drive
sudo cryptsetup luksClose /dev/mapper/backup_drive
```

First, we unmount the volume just like we normally would. Then, we tell `cryptsetup` to close the volume. To mount it again, we would issue the following commands:

```
sudo cryptsetup luksOpen /dev/sdb backup_drive
sudo mount /dev/mapper/backup_drive /media/backup_drive
```

The first of those commands should prompt you for your passphrase. If successful, you can use the second of those commands to mount the volume.

If we wish to change the passphrase, we can use the following command. The disk must not be mounted or open in order for this to work:

```
sudo cryptsetup luksChangeKey /dev/sdb -S 0
```

The command will ask you for the current passphrase, and then the new one twice.

> Keep in mind that you should absolutely be careful typing in the new passphrase, so that you don't lock yourself out of the drive.

That's basically all there is to it. With the `cryptsetup` utility, you can set up your own LUKS-encrypted volumes for storing your most sensitive information. If the disk ever falls into the wrong hands, it won't be as bad a situation as it would have been if the disk had been unencrypted. Breaking a LUKS-encrypted volume would take considerable effort that wouldn't be feasible.

In the next section, we'll explore how we can lock down `sudo`. Since `sudo` is an essential command that gives us the ability to run tasks as other users, we'll want to be sure to lock that down too.

## Locking down sudo

We've been using the `sudo` command throughout the book. In fact, we took a deeper look at it in *Chapter 2, Managing Users and Permissions*. Therefore, I won't go into too much detail regarding `sudo` here, but some things bear repeating as `sudo` has a direct impact on security.

First and foremost, access to sudo should be locked down as much as possible. A user with full sudo access is a threat, plain and simple. All it would take is for someone with full sudo access to make a single mistake with the rm command to cause you to lose data or render your entire server useless. After all, a user with full sudo access can do anything root can do (which is everything).

By default, the user you've created during installation will be made a member of the sudo group. Members of this group have full access to the sudo command. Therefore, you shouldn't make any users a member of this group unless you absolutely have to. In *Chapter 2, Managing Users and Permissions*, I talked about how to control access to sudo with the visudo command; refer to that chapter for a refresher if you need it. In a nutshell, you can lock down access to sudo to specific commands, rather than allowing your users to do everything. For example, if a user needs access to shut down or reboot a server, you can give them access to perform those tasks (and only those tasks) with the following setting:

```
charlie    ALL=(ALL:ALL) /usr/sbin/reboot,/usr/sbin/shutdown
```

For the most part, if a user needs access to sudo, just give them access to the specific commands that are required as part of their job. If a user needs access to work with removable media, give them sudo access for the mount and umount commands. If they need to be able to install new software, give them access to the apt suite of commands, and so on. The fewer permissions you give a user, the better. This goes all the way back to the principle of least privilege that we went over near the beginning of this chapter.

Although most of the information in this section is not new to anyone who has already read *Chapter 2*, *Managing Users and Permissions*, sudo access is one of those things a lot of people don't think about when it comes to security. The sudo command with full access is equivalent to giving someone full access to the entire server. Therefore, it's an important thing to keep in mind when it comes to hardening the security of your network.

## Summary

In this chapter, we looked at the ways in which we can harden the security of our server. A single chapter or book can never give you an all-inclusive list of all the security settings you could possibly configure, but the examples we worked through in this chapter are a great starting point. Along the way, we looked at the concepts of lowering your attack surface, as well as the principle of least privilege. We also looked into securing OpenSSH, which is a common service that many attackers will attempt to use in their favor.

We also looked into Fail2ban, which is a handy daemon that can block other nodes when there are a certain number of authentication failures. We also discussed configuring our firewall, using the UFW utility. Since data theft is also unfortunately common, we covered encrypting our backup disks.

In the next chapter, we'll take a look at troubleshooting our server when things go wrong.

## Further reading

- Landscape documentation: `https://landscape.canonical.com/static/doc/user-guide/`
- Fail2ban manual: `https://www.fail2ban.org/wiki/index.php/MANUAL_0_8`
- `sshd_config` file guide: `https://www.ssh.com/ssh/sshd_config/`
- Ubuntu CVE tracker: `https://people.canonical.com/~ubuntu-security/cve/`
- Password haystacks (find out how secure your password is): `https://www.grc.com/haystack.htm`
- Krebs on security (a great security blog): `https://krebsonsecurity.com/`
- SECURITY NOW (a very informative security podcast): `https://twit.tv/shows/security-now`
- ShieldsUP! (a useful tool to see which ports your router has open): `https://www.grc.com/shieldsup`

# 22
# Troubleshooting Ubuntu Servers

So far, have we've covered many topics surrounding Ubuntu Server, and worked on some really fun projects. We've set up web servers, built automation, and even created infrastructure in the cloud. As the applications and services you've implemented age, your organization may depend on them more and more. But what happens if something your organization relies on suddenly becomes unavailable? What do you do when things don't quite go according to plan?

While it's impossible for us to account for every possible problem that may come up, there are some common places to look for clues when you run into a problem. In this chapter, we'll take a look at some common starting points and techniques that you can utilize when it comes to troubleshooting issues with our servers. Building solid troubleshooting skills is an important focus, and with the concepts explored here, you'll be well on your way.

In this chapter, we will cover:

- Evaluating the scope
- Conducting a root cause analysis
- Viewing system logs
- Tracing network issues
- Troubleshooting resource issues
- Diagnosing defective RAM

The first step with regards to troubleshooting is to analyze the problem and determine how critical a problem it may be. In the next section, we'll explore how to do just that.

## Evaluating the scope

When a problem occurs within your servers or network, your systems will exhibit one or more symptoms. Perhaps an application is much slower than normal, maybe users are unable to access the network, or a server suffers from total failure. There are many problems that can come up at any time, and it can be challenging to keep up.

Once you've identified the symptoms of the problem, the next goal is to identify the overall scope. Essentially, this means determining (as best you can) where the problem is most likely to reside, and how many systems and services are affected. Sometimes the root cause is obvious. For example, if none of your computers are receiving an IP address from your DHCP server, then you'll know straight away to start investigating the logs on that particular server concerning its ability (or inability) to do the job designated for it. In other cases, the cause may not be so obvious. Perhaps you have an application that exhibits problems every now and then but isn't something you can reliably reproduce. In that case, it may take some digging before you know just how large the scope of the problem might be. Sometimes, the culprit is the last thing you expect.

Each component on your network works together with other components, or at least that's how it should be. A network of Linux servers, just as with any other network, is a collection of services (daemons) that compliment and often depend upon one another. For example, DHCP assigns IP addresses to all of your hosts, but it also assigns their default DNS servers as well. If your DNS server has encountered an issue, then your DHCP server would essentially be assigning a non-working DNS server to your clients. Identifying the problem space means that after you identify the symptoms, you'll also work toward reaching an understanding of how each component within your network contributes to, or is affected by, the problem.

With regards to the scope, we identify how far the problem reaches, as well as how many users or systems are affected by the issue. Perhaps just one user is affected, or an entire subnet. This will help you determine the priority of the issue and decide whether this is something essential that you need to fix now, or something that can wait until later. Often, prioritizing is half the battle; sometimes a user will even be under the impression that their issues are more important than others. Use your best judgment.

When identifying the scope, you'll want to answer the following questions as best as you can:

- What are the symptoms of the issue?
- When did this problem first occur?
- Were there any changes made within the network around that time?
- Has this problem happened before? If so, what was done to fix it last time?
- Which servers or nodes are impacted by this issue?
- How many users are impacted?

If the problem is limited to a single machine, then a few really good places to start poking around is to check who is logged in to the server and which commands have recently been entered. Quite often, I've found the culprit just by checking the Bash history for logged on users (or users that have recently logged in). With each user account, there should be a `.bash_history` file in their `home` directory. Within this file is a list of commands that were recently entered. Check this file and see if anyone modified anything recently. I can't tell you how many times this alone has led directly to the answer. And what's even better, sometimes the Bash history leads to the solution. If a problem has occurred before and someone has already fixed it at some point in the past, chances are their efforts were recorded in the Bash history, so you can see what the previous person did to solve the problem just by looking at it. To view the Bash history, you can either view the contents of the `.bash_history` file in a user's `home` directory, or you can simply execute the `history` command as that user.

Additionally, if you check who is currently logged into the server, you may be able to pinpoint if someone is working on an issue already, or perhaps something they're doing caused the issue in the first place. If you enter the `w` command, you can see who is logged in to the server currently. In addition, you'll also see the IP address of the user that's logged in when you run this command. Therefore, if you don't know who corresponds to a user account listed when you run the `w` command, you can check the IP address in your DHCP server to find out who the IP address belongs to, so you can ask that person directly. In a perfect world, other administrators will send out a departmental email when they work on something to make sure everyone is aware. Unfortunately, many don't do this. By checking the logged-in users as well as their Bash history, you're well on your way to determining where the problem originated.

After identifying the problem space and the scope, you can begin narrowing down the issue to help find a cause. Sometimes, the culprit will be obvious. If a website stopped working and you noticed that the Apache configuration on your web server was changed recently, you can attack the problem by investigating the change and who made it.

If the problem is a network issue, such as users not being able to visit websites, the potential problem space is much larger. Your internet gateway may be malfunctioning, your DNS or DHCP server may be down, your internet provider could be having issues, or perhaps your accounting department simply forgot to pay the internet bill. As long as you are able to determine a potential list of targets to focus your troubleshooting on, you're well on your way to finding the issue. As we go through this chapter, I'll talk about some common issues that can come up and how to deal with them.

Understanding the scope of the problem helps us understand just how severe it may be and the number of systems and users impacted, and sometimes investigating the scope can lead you to the root cause of the problem. If you don't already know the underlying cause, you can conduct a root cause of the problem. That's what we'll explore next.

## Conducting a root cause analysis

Once you resolve a problem on your server or network, you'll immediately revel in the awesomeness of your troubleshooting skills. It's a wonderful feeling to have fixed an issue, becoming the hero within your technology department. But you're not done yet. The next step is looking toward preventing this problem from happening again. It's important to look at how the problem started as well as steps you can take in order to help stop the problem from occurring again. This is known as a **root cause analysis**. A root cause analysis may be a report you file with your manager or within your knowledge-base system, or it could just be a memo you document for yourself. Either way, it's an important learning opportunity.

A good root cause analysis has several sides to the equation. First, it will demonstrate the events that led to the problem occurring in the first place. Then, it will contain a list of steps that you've completed to correct the problem. If the problem is something that could potentially recur, you would want to include information about how to prevent it from happening again in the future.

The problem with a root cause analysis is that it's rare that you can be 100 percent accurate. Sometimes, the root cause may be obvious. For example, suppose a user named `Bob` deleted an entire directory that contained files important to your company. If you log into the server and check the logs, you can see that `Bob` not only logged into the server near the time of the incident, but his Bash history literally shows him running the `rm -rf /work/important-files` command. At this point, the case is closed. You figured out how the problem happened, who did it, and you can restore the files from your most recent backup. But a root cause is usually not that cut and dry.

One example I've personally encountered was a pair of **Virtual Machine** (**VM**) servers that were "fencing." At a company I once worked for, our Citrix-based VM servers (which were part of a cluster) both went down at the same time, taking every Linux VM down with them. When I attached a monitor to them, I could see them both rebooting over and over. After I got the servers to settle down, I started to investigate deeper. I read in the documentation for Citrix XenServer that you should never install a cluster of anything less than three machines because it can create a situation exactly like the one I experienced. We only had two servers in that cluster, so I concluded that the servers were set up improperly and the company would need a third server if they wanted to cluster them.

The problem though, is that this root cause analysis wasn't 100 percent perfect. Were the servers having issues because they needed a third server? The documentation did mention that three servers were a minimum, but there's no way to know for sure that was the reason the problem started. However, not only was I not watching the servers when it happened, but I also wasn't the individual who set them up; that person had already left the company. There was no way I could reach an absolute conclusion, but my root cause analysis was sound in the sense that it was the most likely explanation (that we weren't using best practices). Someone could counter my root cause analysis with "but the servers were running fine that way for several years." True, but nothing is absolute when dealing with technology. Sometimes, you never really know. The only thing you can do is make sure everything is set up properly according to the guidelines set forth by the manufacturer.

A good root cause analysis is as sound in logic as it can be, though not necessarily bulletproof. Correlating system events to symptoms is often a good first step but is not necessarily perfect. After investigating the symptoms, solving the issue, and documenting what you've done to rectify it, sometimes the root cause analysis writes itself. Other times, you'll need to read the documentation and ensure that the configuration of the server or daemon that failed was implemented along with best practices. In a worst-case scenario, you won't really know how the problem happened or how to prevent it, but it should still be documented in case other details come to light later. And without documentation, you'll never gain anything from the situation.

A root cause analysis should include details such as the following:

- A description of the issue
- Which application or piece of hardware encountered a fault
- The date and time the issue was first noticed
- What you found while investigating the issue
- What you've done to resolve the issue
- What events, configurations, or faults caused the issue to happen

A root cause analysis should be used as a learning experience. Depending on what the issue was, it may serve as an example of what not to do, or what to do better. In the case of my VM server fiasco, the moral of the story was to follow best practices from Citrix and use three servers for the cluster instead of two. Other times, the end result may be another technician not following proper directives or making a mistake, which is unfortunate. In the future, if the issue were to happen again, you'll be able to look back and remember exactly what happened last time and what you did to fix it. This is valuable, if only because we're all human and prone to forgetting important details after a time. In an organization, a root cause analysis is valuable to show stakeholders that you're able to not only address a problem but are reasonably able to prevent it from happening again.

Often, log files are a great place to find clues, as quite a bit of information surrounding system and application events are stored there. In the next section, we'll explore log files in more detail.

## Viewing system logs

If you're having trouble finding the root cause, or you just want more information regarding a problem that occurred, consider looking through log files. Linux has great logging capabilities, and many of the applications you may be running are writing log files as events happen. If there's an issue, you may be able to find information about it in an application's log file.

Inside the `/var/log` directory, you'll see a handful of logs you can view, which differs from server to server depending on which applications are installed. In quite a few cases, an installed application will create its own log file somewhere within `/var/log`, either in a log file or a log file within a sub-directory of `/var/log`. For example, once you install Apache, it will create log files in the `/var/log/apache2` directory, and looking through those logs may give you a hint as to what may be going on if the problem is related to your web server. These are known as **Application Logs**, which are basically log files created by an application and not the distribution. There are also **System Logs**, which are the log files created by the distribution and allow you to view system events.

Viewing a log file can be done in several ways. One way is to use the `cat` command along with the path and filename of a log file. For example, the Apache access log can be viewed with the following command:

```
cat /var/log/apache2/access.log
```

Some log files are restricted and need `root` privileges in order to access them. If you get a permission denied error when attempting to view a log, use `sudo` in front of any of the commands in this section to view the file.

One problem with the `cat` command is that it will print out the entire file, no matter how big it is. It will scroll by your terminal and if the file is large, you won't be able to see all of it. In addition, if your server is already taxed when it comes to performance, using `cat` can actually tie up the server for a bit in a case where the log file is massive. This will cause you to lose control of your shell until the file stops printing. You can press *Ctrl* + *c* to stop printing the log file, but the server may end up being too busy to respond to *Ctrl* + *c* and show the entire file anyway.

Another method is to use the `tail` command. By default, the `tail` command shows you the last ten lines of a file:

```
tail /var/log/apache2/access.log
```

If you wish to see more than the last ten lines, you can use the `-n` option to specify a different amount. To view the last `100` lines, we would use the following:

```
tail -n 100 /var/log/apache2/access.log
```

Perhaps one of the most useful features of the `tail` command is the `-f` option, which allows you to follow a log file. Basically, this means that as entries are written to the log file, it will scroll by in front of you. It's close to watching the log file in real time:

```
tail -f /var/log/apache2/access.log
```

Once you start using the `follow` option, you'll wonder how you ever lived without it. If you're having a specific problem that you are able to reproduce, you can watch the log file for that application and see the log entries as they appear while you're reproducing the issue. In the case of a DHCP server not providing IP addresses to clients, you can view the output of the `/var/log/syslog` file (the `isc-dhcp-server` daemon doesn't have its own log file), and you can see any errors that come up as your clients try to re-establish their DHCP lease, allowing you to see the problem as it is happening.

Another useful command for viewing logs is `less`. The `less` command allows you to scroll through a log file with the page up and page down keys on your keyboard, which makes it more useful for viewing log files than the `cat` command. You can press *q* to exit the file:

```
less /var/log/apache2/access.log
```

So now that you know a few ways in which you can view these files, which files should you inspect? Unfortunately, there's no one rule, as each application handles its logging differently. Some daemons have their own log file stored somewhere in `/var/log`. Therefore, a good place to check is in that directory, to see if there is a log file with the name of the daemon. Some daemons don't even have their own log file and will use `/var/log/syslog` instead. You may try viewing the contents of the file, while using `grep` to find messages related to the daemon you're troubleshooting. In regard to the `isc-dhcp-server` daemon, the following would narrow down the `syslog` to messages from that specific daemon:

```
cat /var/log/syslog |grep dhcp
```

While troubleshooting security issues, the log file you'll definitely want to look at is the **Authorization Log**, located at `/var/log/auth.log`. You'll need to use the `root` account or `sudo` to view this file. The authorization log includes information regarding authentication attempts to the server, including logins from the server itself, as well as logins over OpenSSH. This is useful for several reasons, among them the fact that if something really bad happens on your server, you can find out who logged in to the server around that time. In addition, if you or one of your users is having trouble accessing the server via OpenSSH, you may want to look at the authorization log for clues, as additional information for OpenSSH failures will be logged there. Often, the `ssh` command may complain about permissions of key files not being correct, which would give you an answer as to why public key authentication stopped working, as OpenSSH expects specific permissions for its files. For example, the private key file (typically `/home/<user>/.ssh/id_rsa`) should not be readable or writable by anyone other than its owning user. You'd see errors within `/var/log/auth.log` mentioning such a thing if that were the case.

Another use case for checking `/var/log/auth.log` is for security, as a high number of login attempts may indicate an intrusion attempt. (Hopefully, you have Fail2ban installed, which we went over in the last chapter.) An unusually high number of failed password attempts may indicate someone trying to log in to the server by brute force. That would definitely be a cause for concern, and you'd want to block their IP address immediately.

The **System Log**, located in `/var/log/syslog`, contains logging information for quite a few different things. It's essentially the Swiss Army knife of Ubuntu's logs. If a daemon doesn't have its own log file, chances are its logs are being written to this file. In addition, information regarding cron jobs will be written here, which makes it a candidate to check when a cron job isn't being executed properly. The `dhclient` daemon, which is responsible for grabbing an IP address from a DHCP server, is also important.

You'll be able to see from `dhclient` events within the system log when an IP address is renewed, and you can also see messages relating to failures if it's not able to obtain an IP address. Also, the `systemd init` daemon itself logs here, which allows you to see messages related to server startup as well as applications it's trying to run.

Another useful log is the `/var/log/dpkg.log` file, which records log entries relating to installing and upgrading packages. If a server starts misbehaving after you roll out updates across your network, you can view this log to see which packages were recently updated. This log will not only give you a list of updated or installed packages, but also a timestamp from when the installation occurred. If a user installed an unauthorized application, you can correlate this log to the authentication log to determine who logged in around that time, and then you can check that user's Bash history to confirm.

Often, log files will get rotated after some time by a utility known as `logrotate`. Inside the `/var/log` directory, you'll see several log files with a `.gz` extension, which means that the original log file was compressed and renamed, and a new log file created in its place. For example, you'll see the syslog file for the system log in the `/var/log` directory, but you'll also see files named with a number and a `.gz` extension as well, such as `syslog.2.gz`. These are compressed logs. Normally, you'd view these logs by uncompressing them and then opening them via any of the methods mentioned in this section. An easier way to do so is with the `zcat` command, which allows you to view compressed files immediately:

```
zcat /var/log/syslog.2.gz
```

There's also `zless`, which serves a similar purpose as the `less` command.

Another useful command for checking logging information is `dmesg`. Unlike other log files, `dmesg` is literally its `own` command. You can execute it from anywhere in the filesystem, and you don't even need `root` privileges to do so. The `dmesg` command allows you to view log entries from the Linux kernel's ring buffer, which can be very useful when troubleshooting hardware issues (such as seeing which disks were recognized by the kernel). When troubleshooting hardware, the system log is also helpful, but using the `dmesg` command may be a good place to check as well.

As I mentioned earlier, on an Ubuntu system there are two types of log files, system logs and application logs. System logs, such as `auth.log` and `dpkg.log`, detail important system events and aren't specific to any one particular application. Application logs become installed when you install their parent package, such as Apache or MariaDB. Application logs create log entries into their own log file.

Some daemons you install will not create their own application log, such as `keepalived` and `isc-dhcp-server`. Since there's no general rule when it comes to which applications log is where, the first step in finding a log file is to see if the application you want log entries from creates its `own` log file. If not, it's likely using a system log.

When faced with a problem, it's important to practice viewing log files at the same time you try and reproduce the problem. Using `follow` mode with `tail` (`tail -f`) works very well for this, as you can watch the log file generate new entries as you try and reproduce the issue. This technique works very well in almost any situation where you're dealing with a misbehaving daemon. This technique can also help narrow down hardware issues. For example, I once dealt with an Ubuntu system where when I plugged in a flash drive, nothing happened. When I followed the log as I inserted and removed the flash drive, I saw the system log update and recognize each insertion and removal. So clearly, the Linux kernel itself saw the hardware and was prepared to use it. This helped me narrow down the problem to being that the desktop environment I was using wasn't updating to show the inserted flash drive, but my hardware and USB ports were operating perfectly fine. With one command, I was able to determine that the issue was a software problem and not related to hardware.

As you can see, Ubuntu contains very helpful log files that will aid you in troubleshooting your servers. Often, when you're faced with a problem, viewing relevant log entries and then conducting a Google search regarding them will result in a useful answer, or at least bring you to a bug report to let you know the problem isn't just limited to you or your configuration. Hopefully, your search results will lead you right to the answer, or at least to a workaround. From there, you can continue to work through the problem until it is solved.

What about network issues? Tracking down the root cause of an issue on the network can be especially challenging, but it's not as difficult as it may seem. In the next section, we'll take a look at a few ways you can trace network issues.

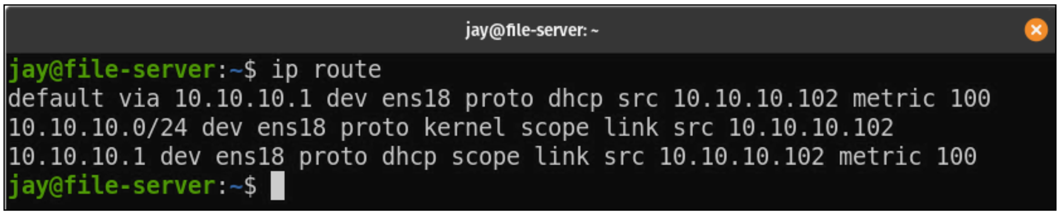
## Tracing network issues

It's amazing how important TCP/IP networking is to the world today. Of all the protocols in use in modern computing, it's by far the most widespread. But it's also one of the most annoying situations to figure out when it's not working well. Thankfully, Ubuntu features really handy utilities you can use in order to pinpoint what's going on.

First, let's look at connectivity. After all, if you can't connect to a network, your server is essentially useless. In most cases, Ubuntu recognizes just about all network cards without fail, and it will automatically connect your server or workstation to your network if it is within reach of a DHCP server.

While troubleshooting, get the obvious stuff out of the way first. The following may seem like a no-brainer, but you'd be surprised how often one can miss something obvious. I'm going to assume you've already checked to make sure network cables are plugged in tight on both ends. Another aspect regarding cabling is that sometimes network cables themselves develop faults and need to be replaced. You should be able to use a cable tester and get a clean signal through the cable.

Routing issues can sometimes be tricky to troubleshoot, but by testing each destination point one by one, you can generally see where the problem lies. Typical symptoms of a routing issue may include being unable to access a device within another subnet, or perhaps not being able to get out to the internet, despite being able to reach internal devices. To investigate a potential routing issue, first check your routing table. You can do so with the `ip route` command. This command will print your current routing table information:



```
jay@file-server:~$ ip route
default via 10.10.10.1 dev ens18 proto dhcp src 10.10.10.102 metric 100
10.10.10.0/24 dev ens18 proto kernel scope link src 10.10.10.102
10.10.10.1 dev ens18 proto dhcp scope link src 10.10.10.102 metric 100
jay@file-server:~$
```

Figure 22.1: Viewing the routing table on an Ubuntu server

In this example, you can see that the default gateway for all traffic is `10.10.10.1`. This is the first entry on the table, which tells us that all traffic to the destination `0.0.0.0` (which is everything) leaves via `10.10.10.1`. As long as ICMP traffic isn't disabled, you should be able to ping this default gateway, and you should be able to ping other nodes within your subnet as well.

To start troubleshooting a routing issue, you would use the information shown after printing your routing table to conduct several ping tests. First, try to ping your default gateway. If you cannot, then you've found the issue. If you can, try running the `traceroute` command. This command isn't available by default, but all you'll have to do is install the `traceroute` package, so hopefully you have it installed on the server. If you do, you can run `traceroute` against a host, such as an external URL, to find out where the connection drops. The `traceroute` command should show every hop between you and your target. Each "hop" is basically another default gateway. You traverse through one gateway after another until you ultimately reach your destination. With the `traceroute` command, you can see where the chain stops. In all likelihood, you'll find that perhaps the problem isn't even on your network, but perhaps your internet service provider is where the connection drops.

DNS issues don't happen very often, but by using a few tricks, you should be able to resolve them. Symptoms of DNS failures will usually result in a host being unable to access internal or external resources by name.

Knowing whether the problem is with internal or external hosts (or both) should help you determine whether it's your DNS server that's the problem, or perhaps the DNS server at your ISP.

The first step in pinpointing the source of DNS woes is to ping a known IP address on your network, preferably the default gateway. If you can ping it, but you can't ping the gateway by name, then you probably have a DNS issue. You can confirm a potential DNS issue by using the `nslookup` command against the domain, such as this:

```
nslookup myserver.local
```

In addition, make sure you try and ping external resources as well, such as a website. This will help you narrow down the scope of the issue.

You will also want to know which DNS server your host is sending queries to. In the past, finding out which DNS server is assigned to your host was as simple as inspecting the contents of `/etc/resolv.conf`. However, nowadays this file will often refer to a local resolver instead and won't reveal the actual server requests are being sent to. To find out the real DNS server that's assigned to your host, the following command will do the trick:

```
systemd-resolve --status | grep DNS\ Servers
```

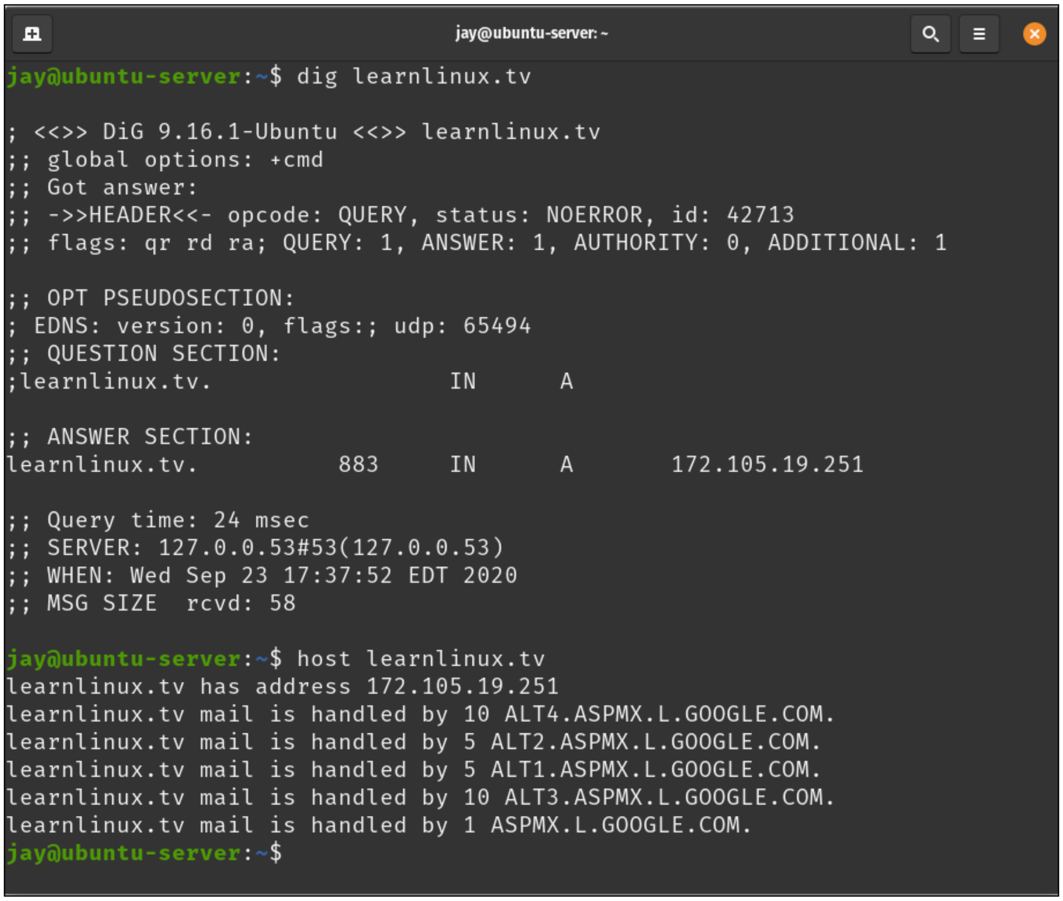
Are they what you expect? If not, you can temporarily fix this problem by removing the incorrect name server entries from this file and replacing them with the correct IP addresses. The reason I suggest this as a temporary fix and not a permanent one is because the next thing you'll need to do is investigate how the invalid IP addresses got there in the first place. Normally, these are assigned by your DHCP server. As long as your DHCP server is sending out the appropriate name server list, you shouldn't run into this problem. If you're using a static IP address, then perhaps there's an error in your Netplan config file.

A useful method of pinpointing DNS issues in regard to being unable to resolve external sites is to temporarily switch your DNS provider on your local machine. Normally, your machine is going to use your external DNS provider, such as the one that comes from your ISP. Your external DNS server is something we went through setting up in *Chapter 11, Setting up Network Services*, specifically the forwarders section of the configuration for the `bind9` daemon. The forwarders used by the `bind9` daemon is where it sends traffic if it isn't able to resolve your request based on its internal list of hosts.

You could consider bypassing this by changing your local workstation's DNS name servers to Google's, which are `8.8.8.8` and `8.8.4.4`. If you're able to reach the external resource after switching your name servers, you can be reasonably confident that your forwarders are the culprit.

I've actually seen situations in which a website has changed its IP address, but the ISP's DNS servers didn't get updated quickly enough, causing some clients to be unable to reach a site they need to perform their job. Switching everyone to alternate name servers (by adjusting the `forwarders` option, as we did in *Chapter 11, Setting up Network Services*) was the easiest way they could work around the issue.

Some additional tools to consider while checking your server's ability to resolve DNS entries are `dig` and `nslookup`. You should be able to use both commands to test your server's DNS settings. Both commands are used with a host name or domain name as an option. The `dig` command will present you with information regarding the address (`A`) record of the DNS zone file responsible for the IP address or domain. The `host` command should return the IP address of the host you're trying to reach. Here's some example output:



```
jay@ubuntu-server:~$ dig learnlinux.tv

; <<>> DiG 9.16.1-Ubuntu <<>> learnlinux.tv
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 42713
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;learnlinux.tv.                 IN      A

;; ANSWER SECTION:
learnlinux.tv.          883     IN      A       172.105.19.251

;; Query time: 24 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
;; WHEN: Wed Sep 23 17:37:52 EDT 2020
;; MSG SIZE  rcvd: 58

jay@ubuntu-server:~$ host learnlinux.tv
learnlinux.tv has address 172.105.19.251
learnlinux.tv mail is handled by 10 ALT4.ASPMX.L.GOOGLE.COM.
learnlinux.tv mail is handled by 5 ALT2.ASPMX.L.GOOGLE.COM.
learnlinux.tv mail is handled by 5 ALT1.ASPMX.L.GOOGLE.COM.
learnlinux.tv mail is handled by 10 ALT3.ASPMX.L.GOOGLE.COM.
learnlinux.tv mail is handled by 1 ASPMX.L.GOOGLE.COM.
jay@ubuntu-server:~$
```

Figure 22.2: Output of the dig and host commands

Hardware support is also critical when it comes to networking. If the Linux kernel doesn't support your network hardware, then you'll likely run into a situation where the distribution doesn't recognize or do anything when you insert a network cable, or in the case of wireless networking, doesn't show any nearby networks despite there being one or more. Unlike the Windows platform, hardware support is generally baked right into the kernel when it comes to Linux. While there are exceptions to this, the Linux kernel shipped with a distribution typically supports hardware the same age as itself or older. In the case of Ubuntu 20.04 LTS (which was released in April of 2020), it's able to support hardware released as of the beginning of 2020 and older. Future releases of Ubuntu Server will publish hardware enablement updates, which will allow Ubuntu Server 20.04 to support newer hardware and chip-sets once they come out. Typically, Ubuntu will release several point releases during the life of a supported distribution, such as 20.04.1, 20.04.2, and so on. As long as you're using the latest one, you'll have access to the latest hardware support that Ubuntu has made available at the time.

In other cases, hardware support may depend on external kernel modules. In the case of a missing hardware driver, the first thing you should try when faced with network hardware that's not recognized is to look up the hardware using a search engine, typically the search terms `<hardware name> Ubuntu` will do the trick. But what do you search for? To find out the hardware string for your network device, try the `lspci` command:

```
lspci | grep -i net
```

The `lspci` command lists hardware connected to your server's PCI bus. Here, we're using the command with a case insensitive `grep` search for the word `net`:

```
lspci |grep -i net
```

This should return a list of networking components available in your server. On my machine, for example, I get the following output:

```
01:00.1 Ethernet controller: Realtek Semiconductor Co., Ltd.
RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller (rev 12)

02:00.0 Network controller: Intel Corporation Wireless 8260 (rev 3a)
```

As you can see, I have a wired and wireless network card on this machine. If one of them wasn't working, I could search online for information by searching for the hardware string and the keyword `Ubuntu`, which should give me results pertaining to my exact hardware. If a package is required to be installed, the search results will likely give me some clues as to what package I need to install.

Without having network access though, the worst-case scenario is that I may have to download the package from another computer and transfer it to the server via a flash drive. That's certainly not a fun thing to need to do, but it does work if the latest Ubuntu installation media doesn't yet offer full support for your hardware.

Another potential problem point is DHCP. When it works well, DHCP is a wonderfully magical thing. When it stops working, it can be frustrating. But generally, DHCP issues often end up being a lack of available IP addresses, the DHCP daemon (`isc-dhcp-server`) not running, an invalid configuration, or hosts that have clocks that are out of sync (all servers should have the `ntp` package installed).

If you have a server that is unable to obtain an IP address via DHCP and your network utilizes a Linux-based DHCP server, check the system log (`/var/log/syslog`) for events related to `dhcpd`. Unfortunately, there's no command you can run that I've ever been able to find that will print how many IP address leases your DHCP server has remaining, but if you run out, chances are you'll see log entries related to an exhausted pool in the system log. In addition, the system log will also show you attempts from your nodes to obtain an IP address as they attempt to do so. Feel free to use `tail -f` against the system log, to watch for any events related to DHCP leases.

In some cases, a lack of DHCP leases being available can come down to having a very generous lease time enabled. Some administrators will give their clients up to a week for the lease time, which is generally unnecessary. A lease time of one day is fine for most networks, but ultimately the lease time you decide on is up to you. In *Chapter 11*, *Setting up Network Services*, we looked at configuring our DHCP server, so feel free to refer to that chapter if you need a refresher on how to configure the `isc-dhcp-server` daemon.

Although it's probably not the first thing you'll think of while facing DHCP issues, hosts having out-of-sync clocks can actually contribute to the problem. DHCP requests are `timestamped` on both the client and the server, so if the clock is off by a large degree on one, the timestamps will be off as well, causing the DHCP server to become confused. Surprisingly, I've seen this come up fairly often. I recommend standardizing NTP across your network as early on as you can. DHCP isn't the only service that suffers when clocks are out of sync, file synchronization utilities also require accurate time. If you ensure NTP is installed on all of your clients and it's up to date and working, you should be in good shape. Using configuration management utilities such as Ansible to ensure NTP is not only configured but is running properly on all the machines in your network will only benefit you.

Of course, there are many things that can go wrong when it comes to networking, but the information here should cover the majority of issues. In summary, troubleshooting network issues generally revolves around ping tests. Trying to ping your default gateway, tracing failed endpoints with `traceroute`, and troubleshooting DNS and DHCP will take care of a majority of issues. Then again, faulty hardware such as failed network cards and bad cabling will no doubt present themselves as well.

Our servers utilize storage, CPU, memory, and other resources to provide us with value and serve our clients. In the next section, we'll take a closer look at how to check those resources.

## Troubleshooting resource issues

I don't know about others, but it seems that a majority of my time troubleshooting servers is usually spent pinpointing resource issues. By resources, I'm referring to CPU, memory, disk, input/output, and so on. Generally, issues come down to a user storing too many large files, a process going haywire that consumes a large amount of CPU, or a server running out of memory. In this section, we'll go through some of the common things you're likely to run into while administering Ubuntu servers.

First, let's revisit topics related to storage. In *Chapter 9, Managing Storage Volumes*, we went over concepts related to this already, and many of those concepts also apply to troubleshooting as well. Therefore, I won't spend too much time on those concepts here, but it's worth a refresher in regard to troubleshooting storage issues. First, whenever you have users that are complaining about being unable to write new files to the server, the following two commands are the first you should run. You are probably already well aware of these, but they're worth repeating:

```
df -h
df -i
```

The first `df` command variation gives you information regarding how much space is used on a drive, in a human-readable format (the `-h` option), which will print the information in terms of megabytes and gigabytes. The `-i` option in the second command gives you information regarding in-use and available inodes. The reason you should also run this is that on a Linux system, it can report storage as full even if there's plenty of free space. But if there are no remaining inodes, it's the same as being full, but the first command wouldn't show the usage as 100 percent when no inodes are free. Usually, the number of inodes a storage medium has available is extremely generous, and the limit is hard to hit. However, if a service is creating new log files over and over every second, or a mail daemon grows out of control and generates a huge backlog of undelivered mail, you'd be surprised how quickly inodes can empty out.

Of course, once you figure out that you have an issue with full storage, the next logical question becomes, what is eating up all my free space? The `df` commands will give you a list of storage volumes and their sizes, which will tell you at least which disk or partition to focus your attention on. My favorite command for pinpointing storage hogs, as I mentioned in *Chapter 9, Managing Storage Volumes*, is the `ncdu` command. While not installed by default, `ncdu` is a wonderful utility for checking to see where your storage is being consumed the most. If run by itself, `ncdu` will scan your server's entire filesystem. Instead, I recommend running it with the `-x` option, which will limit it to a specific folder as a starting point. For example, if the `/home` partition is full on your server, you might want to run the following to find out which directory is using the most space:

```
sudo ncdu -x /home
```

The `-x` option will cause `ncdu` to not cross filesystems. This means if you have another disk mounted within the folder you're scanning, it won't touch it. With `-x`, `ncdu` is only concerned with the target you give it.

If you aren't able to utilize `ncdu`, there's also the `du` command, which takes some extra work. The `du -h` command, for example, will give you the current usage of your current working directory, with human-readable numbers. It doesn't traverse directory trees by default like `ncdu` does, so you'd need to run it on each sub-directory until you manually find the directory that's holding the most files. A very useful variation of the `du` command, nicknamed `ducks`, is the following. It will show you the top 15 largest directories in your current working directory:

```
du -cksh * | sort -hr | head -n 15
```

Another issue with storage volumes that can arise is to do with filesystem integrity. Most of the time, these issues only seem to come up when there's an issue with power, such as a server powering off unexpectedly. Depending on the server and the formatting you've used when setting up your storage volumes (and several other factors), power issues are handled differently from one installation to another. In most cases, a filesystem check (`fsck`) will happen automatically during the next boot. If it doesn't, and you're having odd issues with storage that can't be explained, a manual filesystem check is recommended. Scheduling a filesystem check is actually very easy:

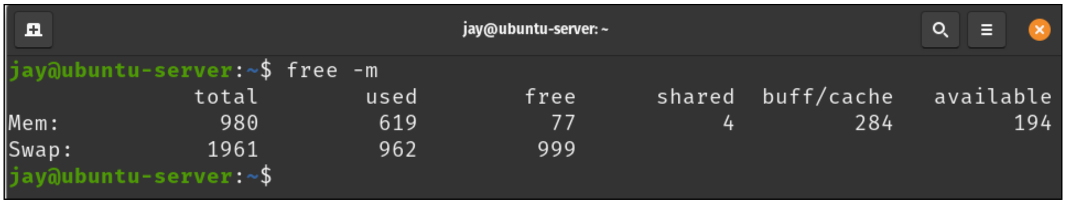
```
sudo touch /forcefsck
```

The previous command will create an empty file, `forcefsck`, at the root of the filesystem. When the server reboots and it sees this file, it will trigger a filesystem check on that volume and then remove the file. If you'd like to check a filesystem other than the root volume, you can create the `forcefsck` file elsewhere. For example, if your server has a separate `/home` partition, you could create the file there instead to check that volume:

```
sudo touch /home/forcefsck
```

The filesystem check will usually complete fairly quickly, unless there's an issue it needs to fix. Depending on the nature of the problem, the issue could be repaired quickly or perhaps it will take a while. I've seen some really bad integrity issues that have taken over four hours to fix, but I've seen others fixed in a matter of seconds. Sometimes it will finish so quickly that it will scroll by so fast during boot that you may miss seeing it. In case of a large volume, you may want to schedule the `fsck` check to happen after-hours in case the scan takes a long time.

With regards to issues with memory, the `free -m` command will give you an overview of how much memory and swap is available on your server. It won't tell you what exactly is using up all your memory, but you'll use it to see if you're in jeopardy of running out. The **free** column from the output of the `free` command will show you how much memory is remaining, and allow you to make a decision on when to take action:



```
jay@ubuntu-server:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:            980         619          77           4         284         194
Swap:          1961         962         999
jay@ubuntu-server:~$
```

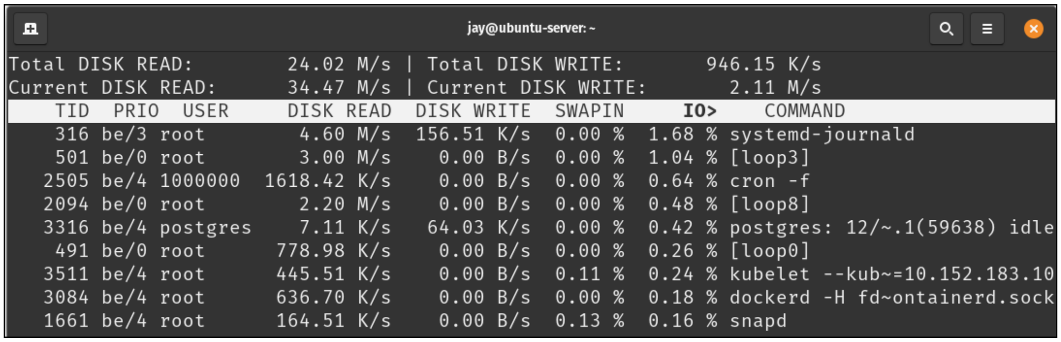
Figure 22.3: Checking the available memory on an Ubuntu server

In *Chapter 8, Monitoring System Resources*, we took a look at the `htop` command, which helps us answer the question of "what" is using up our resources. Using `htop` (once installed), you can sort the list of processes by CPU or memory usage by pressing *F6* and then selecting a new sort field, such as `PERCENT_CPU` or `PERCENT_MEM`. This will give you an idea of what is consuming resources on your server, allowing you to make a decision on what to do about it. The action you take will differ from one process to another, and your solution may range from adding more memory to the server to tuning the application to have a lower memory ceiling. But what do you do when the results from `htop` don't correlate to the usage you're seeing? For example, what if your load average is high, but no process seems to be consuming a large portion of CPU?

One command I haven't discussed so far in this book is `iotop`. While not installed by default, the `iotop` utility is definitely a must-have, so I recommend you install the `iotop` package. The `iotop` utility itself needs to be run as `root` or with `sudo`:

```
sudo iotop
```

The `iotop` command will allow you to see how much data is being written to or read from your disks. **Input/Output** (IO) definitely contributes to a system's load, and not all resource monitoring utilities will show this usage. If you see a high load average but nothing in your resource monitor shows anything to account for it, check the IO. The `iotop` utility is a great way to do that, as if data is bottle-necked while being written to disk, that can account for a serious overhead in IO that will slow other processes down. If nothing else, it will give you an idea of which process is misbehaving, in case you need to kill it:



```
Total DISK READ:        24.02 M/s | Total DISK WRITE:       946.15 K/s
Current DISK READ:      34.47 M/s | Current DISK WRITE:       2.11 M/s
    TID  PRIO  USER       DISK READ  DISK WRITE  SWAPIN      IO>    COMMAND
    316 be/3 root         4.60 M/s  156.51 K/s  0.00 %  1.68 % systemd-journald
    501 be/0 root         3.00 M/s    0.00 B/s  0.00 %  1.04 % [loop3]
   2505 be/4 1000000   1618.42 K/s    0.00 B/s  0.00 %  0.64 % cron -f
   2094 be/0 root         2.20 M/s    0.00 B/s  0.00 %  0.48 % [loop8]
   3316 be/4 postgres     7.11 K/s   64.03 K/s  0.00 %  0.42 % postgres: 12/~.1(59638) idle
    491 be/0 root       778.98 K/s    0.00 B/s  0.00 %  0.26 % [loop0]
   3511 be/4 root       445.51 K/s    0.00 B/s  0.11 %  0.24 % kubelet --kub~=10.152.183.10
   3084 be/4 root       636.70 K/s    0.00 B/s  0.00 %  0.18 % dockerd -H fd~ontainerd.sock
   1661 be/4 root       164.51 K/s    0.00 B/s  0.13 %  0.16 % snapd
```

Figure 22.4: The iotop utility running on an Ubuntu server

The `iotop` window will refresh on its own, sorting processes by the column that is highlighted. To change the highlight, you'll only need to press the left and right arrows on your keyboard. You can sort processes by columns such as `IO`, `SWAPIN`, `DISK WRITE`, `DISK READ`, and others. When you're finished with the application, press *q* to quit.

The utilities we looked at in this section are very useful when identifying issues with bottle-necked resources. What you do to correct the situation after you find the culprit will depend on the daemon. Perhaps there's an invalid configuration, or the daemon has encountered a fault and needs to be restarted. Often, checking the logs may lead you to an answer as to why a daemon misbehaves. In the case of full storage, almost nothing beats `ncdu`, which will almost always lead you directly to the problem. Tools such as `htop` and `iotop` allow you to view additional information regarding resource usage as well, and `htop` even allows you to kill a misbehaving process right from within the application, by pressing *F9*.

What do you do when system memory (RAM) becomes physically defective? It happens more often than you'd think. In the next section, we'll look at a way we can test our RAM to see if it's defective or not.

## Diagnosing defective RAM

All server and computing components can and will fail eventually, but there are a few pieces of hardware that seem to fail more often than others. Fans, power supplies, and hard disks definitely make the list of common things administrators will end up replacing, but defective memory is also a situation I'm sure you'll run into eventually.

Although memory sticks becoming defective is something that could happen, I made it the last section in this chapter because unfortunately I can't give you a definitive list of symptoms to look out for that indicate that memory is the source of an issue. RAM issues are very mysterious in nature, and each time I've run into one, I've always stumbled across memory being bad only after troubleshooting everything else. It's for this reason that nowadays I'll often test the memory on a server or workstation first, since it's very easy to do. Even if memory has nothing to do with an issue, it's worth checking anyway since it could become a problem later.

Most distributions of Linux (Ubuntu included) feature **Memtest86+** right on the installation media. Whether you create a bootable CD or flash drive, there's a memory test option available from the Ubuntu Server media. When you first boot from the Ubuntu Server media, you'll see an icon toward the bottom indicating that you can press a key to bring up a menu (if you don't press a key, the installer will automatically start). Next, you'll be asked to choose your language, and then you'll be shown an installation menu. Among the choices there will be a **Test memory** option:

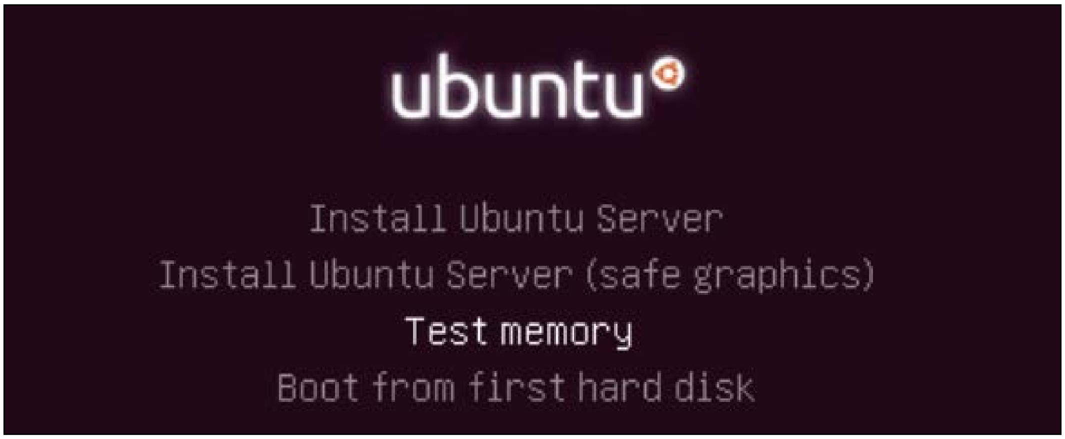

ubuntu
Install Ubuntu Server
Install Ubuntu Server (safe graphics)
Test memory
Boot from first hard disk

Figure 22.5: The main menu of the Ubuntu installer, showing a Test memory option

Other editions of Ubuntu, such as the Ubuntu desktop distribution or any of its derivatives, also feature an option to test memory. Even if you don't have installation media handy for the server edition, you can use whichever version you have. From one distribution or edition to another, the **Memtest86+** program doesn't change.

When you choose the **Test memory** option from your installation media, the **Memtest86+** program will immediately get to work and start testing your memory (press *Esc* to exit the test). The test may take a long time, depending on how much memory your workstation or server has installed. It can take minutes or even hours to complete. Generally speaking, when your machine has defective RAM, you'll see a bunch of errors show up relatively quickly, usually within the first 5-10 minutes. If you don't see errors within 15 minutes, you're most likely in good shape. In my experience, every time I've run into defective memory, I'll see errors in 15 minutes or less (usually within 5). Theoretically, though, you could very well have a small issue with your memory modules that may not show up until after 15 minutes, so you should let the test finish if you can spare the time for it.

The main question becomes when to run **Memtest86+** on a machine. In my experience, symptoms of bad memory are almost never the same from one machine to another. Usually, you'll run into a situation where a server doesn't boot properly, applications close unexpectedly, applications don't start at all, or perhaps an application is behaving irregularly. In my view, testing memory should be done whenever you experience a problem that doesn't necessarily seem straightforward. In addition, you may want to consider testing the memory on your server before you roll it out into production. That way, you can ensure that it starts out as free of hardware issues as possible. If you install new memory modules, make sure to test the RAM right away.

If the test does report errors, you'll next want to find out which memory module is faulty. This can be difficult, as some servers can have more than a dozen memory modules installed. To narrow it down, you'd want to test each memory module independently if you can, until you find out which one is defective. You should also continue to test the other modules, even after you discover the culprit. The reason for this is that having multiple memory modules going bad isn't outside the realm of possibility, considering whatever situation led to the first module becoming defective may have affected others.

Another tip I'd like to pass along regarding memory is that when you do discover a bad stick of memory, it's best to erase the hard disk and start over if you can. I understand that this isn't always feasible, and you could have many hours logged into setting up a server. Some servers can take weeks to rebuild, depending on their workload. But at least keep in mind that any data that passes through defective RAM can become corrupted.

This means that data at rest (data stored on your hard disk) may be corrupt if it was sitting in a defective area of RAM before it was written to disk. When a server or workstation encounters defective RAM, you really can't trust it anymore. I'll leave the decision on how to handle this situation up to you (hopefully you'll never encounter it at all), but just keep this in mind as you plan your course of action. Personally, I don't trust an installation of any operating system after its hardware has encountered such issues.

I also recommend that you check the capacitors on your server's motherboard whenever you're having odd issues. Although this isn't necessarily related to memory, I mention it here because the symptoms are basically the same as bad memory when you have bad capacitors. I'm not asking you to get a voltage meter or do any kind of electrician work, but sometimes it may make sense to open the case of your server, shine a flashlight on the capacitors, and see if any of them appear to be leaking fluid or expanding. The reason I bring this up is that I've personally spent hours troubleshooting a machine (more than once) where I would test the memory and hard disk, and look through system logs, without finding any obvious causes, only to later look at the hardware and discover that capacitors on the motherboard were leaking. It would have saved me a lot of time if I had simply looked at the capacitors. And that's really all you have to do: just take a quick glance around the motherboard and look for anything that doesn't seem right.

## Summary

While Ubuntu is generally a very stable and secure platform, it's important to be prepared for problems occurring and to know how to deal with them. In this chapter, we discussed common troubleshooting we can perform when our servers stop behaving themselves. We started off by evaluating the scope, which gives us an understanding of how many users or servers are affected by the issue. Then, we looked into Ubuntu's log files, which are a treasure trove of information that we can use to pinpoint issues and narrow down the problem. We also covered several networking issues that can come up, such as issues with DHCP, DNS, and routing. We certainly can't predict problems before they occur, nor can we be prepared in advance for every type of problem that can possibly happen. However, applying sound logic and common sense to problems will go a long way in helping us figure out the root cause.

In the next chapter, we will take a look at preventing disasters in the first place and recovering from them if they happen anyway. See you there!

## Further reading

- Reporting Bugs in Ubuntu Server: `https://ubuntu.com/server/docs/reporting-bugs`

# 23 Preventing Disasters

In an enterprise network, a disaster can strike at any time. While, as administrators, we always do our best to design the most stable and fault-tolerant server implementations we possibly can, what matters most is how we are able to deal with disasters when they do happen. As stable as server hardware is, any component of a server can fail at any time. In the face of a disaster, we need a plan. How can you attempt to recover data from a failed disk? What do you do when your server all of a sudden decides it doesn't want to boot? These are just some of the questions we'll answer as we take a look at several ways we can prevent and recover from disasters. In our final chapter, we'll cover the following topics:

- Preventing disasters
- Utilizing Git for configuration management
- Implementing a backup plan
- Replacing failed RAID disks
- Utilizing bootable recovery media

We'll start off our final chapter by looking at a few tips to help prevent disaster.

## Preventing disasters

As we proceed through this chapter, we'll look at ways we can recover from disasters. However, if we can prevent a disaster from occurring in the first place, then that's even better. We certainly can't prevent every type of disaster that could possibly happen but having a good plan in place and following that plan will lessen the likelihood. A good disaster recovery plan will include a list of guidelines to be followed with regard to implementing new servers and managing current ones.

This plan may include information such as an approved list of hardware (such as hardware configurations known to work efficiently in an environment), as well as rules and regulations for users, a list of guidelines to ensure physical and software security, proper training for end users, and method change control. Some of these concepts we've touched on earlier in the book but are worth repeating from the standpoint of disaster prevention.

First, we talked about the **principle of least privilege** back in *Chapter 21, Securing Your Server*. The idea is to give your users as few permissions as possible. This is very important for security, as you want to ensure only those trained in their specific jobs are able to access and modify only the resources that they are required to. Accidental data deletion happens all the time. To take full advantage of this principle, create a set of groups as part of your overall security design. List departments and positions in your company and the types of activities each is required to perform. Create system groups that correspond to those activities. For example, create an accounting-ro and accounting-rw group to categorize users within your Accounting department that should have the ability to only read or read and write data. If you're simply managing a home file server, be careful of open network shares where users have read and write access by default. By allowing users to do as little as possible, you'll prevent a great many disasters right away.

In *Chapter 2, Managing Users and Permissions* (as well as *Chapter 21, Securing Your Server*), we talked about best practices for the `sudo` command. While the `sudo` command is useful, it's often misused. By default, anyone that's a member of the `sudo` group can use `sudo` to do whatever they want. We talked about how to restrict `sudo` access to particular commands, which is always recommended. Only trusted administrators should have full access to `sudo`. Everyone else should have `sudo` permissions only if they really need them, and even then, only when it comes to commands that are required for their job. A user with full access to `sudo` can delete an entire filesystem, so it should never be taken lightly.

In regard to network shares, it's always best to default to read-only whenever possible. This isn't just because of the possibility of a user accidentally deleting data; it's always possible for applications to malfunction and delete data as well. With a read-only share, the modification or deletion of files isn't possible. Additional read-write shares can be created for those who need it, but if possible, always default to read-only.

Although I've spent a lot of time discussing security in a software sense, physical security is important too. For the purposes of this book, physical security doesn't really enter the discussion much because our topic is specifically Ubuntu Server, and nothing you install on Ubuntu is going to increase the physical security of your servers. It's worth noting, however, that physical security is every bit as important as securing your operating systems, applications, and data files.

All it would take is someone tripping over a network cable in a server room to disrupt an entire subnet or cause a production application to go offline. Server rooms should be locked, and only trusted administrators should be allowed to access your equipment. I'm sure this goes without saying and may sound obvious, but I've worked at several companies that did not secure their server room. Nothing good ever comes from placing important equipment within arm's reach of unauthorized individuals.

In this section, I've mentioned *Chapter 21*, *Securing Your Server*, a couple of times. A good majority of a disaster prevention plan includes a focus on security. This includes, but is not limited to, ensuring security updates are installed in a timely fashion, utilizing security applications such as failure monitors and firewalls, and ensuring secure settings for OpenSSH. I won't go over these concepts again here since we've already covered them, but essentially security is a very important part of a disaster prevention plan. After all, users cannot break what they cannot access, and hackers will have a harder time penetrating your network if you designed it in a security-conscious way.

Effective disaster prevention consists of a list of guidelines for things such as user management, server management, application installations, security, and procedure documents. A full walkthrough of proper disaster prevention would be an entire book in and of itself. My goal with this section is to provide you with some ideas you can use to begin developing your own plan. A disaster prevention plan is not something you'll create all at once but is rather something you'll create and refine indefinitely as you learn more about security and what types of things to watch out for.

The configuration files on your server determine the behavior of your services and applications, and backing up can enable you to recover their state. When it comes to managing the state of files, Git is a very powerful tool to do just that, and we'll talk about it next.

## Utilizing Git for configuration management

One of the most valuable assets on a server is its configuration. This is second only to the data the server stores. Often, when we implement a new technology on a server, we'll spend a great deal of time editing configuration files all over the server to make it work as best as we can. This can include any number of things, from Apache virtual host files to DHCP server configuration, DNS zone files, and more. If a server were to encounter a disaster from which the only recourse was to completely rebuild it, the last thing we'd want to do is re-engineer all of this configuration from scratch. This is where Git comes in.

In a typical development environment, an application being developed by a team of engineers can be managed by Git, each contributing to a repository that hosts the source code for their software. One of the things that makes Git so useful is how you're able to go back to previous versions of a file in an instant, as it keeps a history of all changes made to the files within the repository.

Git isn't just useful for software engineers, though. It's also a really useful tool we can leverage for keeping track of configuration files on our servers. For our use case, we can use it to record changes to configuration files and push them to a central server for backup. When we make configuration changes, we push the changes back to our Git server. If for some reason we need to restore the configuration after a server fails, we can simply download our configuration files from Git back onto our new server. Another useful aspect of this approach is that if an administrator implements a change to a configuration file that breaks a service, we can simply revert to a known working commit and we'll be immediately back up and running. You can even correlate changes in log files to changes made at around the same time in a Git repository, which makes it easier to narrow down the root cause of an issue.

Configuration management on servers is so important, in fact, I highly recommend that every Linux administrator takes advantage of version control for this purpose. Although it may seem a bit tricky at first, it's actually really easy to get going once you practice with it. Once you've implemented Git for keeping track of all your server's configuration files, you'll wonder how you ever lived without it. We covered Git briefly in *Chapter 15, Automating Server Configuration with Ansible,* where I walked you through creating a repository on GitHub to host Ansible configuration. However, GitHub may or may not be a good place for your company's configuration, because not only do most companies have policies against sharing internal configuration, you may have sensitive information that you wouldn't want others to see. Thankfully, you don't really need GitHub in order to use Git; you can use a local server for Git on your network.

I'll walk you through what you'll need to do in order to implement this approach. To get started, you'll want to install the `git` package:

```
sudo apt install git
```

In regard to your Git server, you don't necessarily have to dedicate a server just for this purpose; you can use an existing server. The only important aspect here is that you have a central server onto which you can store your Git repositories. All your other servers will need to be able to reach it via your network. On whatever machine you've designated as your Git server, install the `git` package as well. Believe it or not, that's all there is to it. Since Git uses OpenSSH by default, we only need to make sure the `git` package is installed on the server as well as our clients.

We'll need a directory on that server to house our Git repositories, and the users on your servers that utilize Git will need to be able to modify that directory.

Now, think of a configuration directory that's important to you, that you want to place into version control. A good example is the `/etc/apache2` directory on a web server. That's what I'll use in my examples in this section. But you're certainly not limited to that. Any configuration directory you would rather not lose is a good candidate. If you choose to use a different configuration path, change the paths I give you in my examples to that path.

On the server, create a directory to host your repositories. I'll use `/git` in my examples:

```
sudo mkdir /git
```

Next, you'll want to modify this directory to be owned by the administrative user you use on your Ubuntu servers. Typically, this is the user that was created during the installation of the distribution. You can use any user you want actually, just make sure this user is allowed to use OpenSSH to access your Git server. Change the ownership of the `/git` directory so it is owned by this user. My user on my Git server is `jay`, so in my case, I would change the ownership with the following command:

```
sudo chown jay:jay /git
```

Next, we'll create our Git repository within the `/git` directory. For Apache, I'll create a bare repository for it within the `/git` directory. A bare repository is basically a skeleton of a Git repository that doesn't contain any useful data, just some default configuration to allow it to act as a Git folder. To create the bare repository, `cd` into the `/git` directory and execute:

```
git init --bare apache2
```

You should see the following output:

```
Initialized empty Git repository in /git/apache2/
```

That's all we need to do on the server for now for the purposes of our Apache repository. On your client (the server that houses the configuration you want to place under version control), we'll copy this bare repository by cloning it. To set that up, create a `/git` directory on your Apache server (or whatever kind of server you're backing up) just as we did before. Then, `cd` into that directory and clone your repository with the following command:

```
git clone 192.168.1.101:/git/apache2
```

For that command, replace the IP address with either the IP address of your Git server or its hostname if you've created a DNS entry for it. You should see the following output, warning us that we've cloned an empty repository:

```
warning: You appear to have cloned an empty repository
```

This is fine, we haven't actually added anything to our repository yet. If you were to `cd` into the directory we just cloned and list its storage, you'd see it as an empty directory. If you use `ls -a` to view hidden directories as well, you'll see a `.git` directory inside. Inside the `.git` directory, we'll have configuration items for Git that allow this repository to function properly. For example, the config file in the `.git` directory contains information on where the remote server is located. We won't be manipulating this directory; I just wanted to give you a quick overview of what its purpose is.

Note that if you delete the `.git` directory in your cloned repository, that basically removes version control from the directory and makes it a normal directory.

Anyway, let's continue. We should first make a backup of our current `/etc/apache2` directory on our web server, in case we make a mistake while converting it to being version controlled:

```
sudo cp -rp /etc/apache2 /etc/apache2.bak
```

Then, we can move all the contents of `/etc/apache2` into our repository:

```
sudo mv /etc/apache2/* /git/apache2
```

The `/etc/apache2` directory is now empty. Be careful not to restart Apache at this point; it won't see its configuration files and will fail. Remove the (now empty) `/etc/apache2` directory:

```
sudo rm /etc/apache2
```

Now, let's make sure that Apache's files are owned by `root`. The problem though is if we use the `chown` command, as we normally would to change ownership, we'll also change the `.git` directory to be owned by `root` as well. We don't want that, because the user responsible for pushing changes should be the owner of the `.git` folder. The following command will change the ownership of the files to `root`, but won't touch hidden directories such as `.git`:

```
sudo find /git/apache2 -name '.?*' -prune -o -exec chown root:root {} +
```

When you list the contents of your repository directory now, you should see that all files are owned by `root`, except for the `.git` directory, which should be owned by your administrative user account.

Next, create a symbolic link to your Git repository so the `apache2` daemon can find it:

```
sudo ln -s /git/apache2 /etc/apache2
```

At this point, you should see a symbolic link for Apache, located at `/etc/apache2`. If you list the contents of `/etc` while grepping for `apache2`, you should see it as a symbolic link:

```
ls -l /etc | grep apache2
```

The directory listing will look similar to the following:

```
lrwxrwxrwx 1 root root 37 2020-06-25 20:59 apache2 -> /git/apache2
```

If you reload Apache, nothing should change and it should find the same configuration files as it did before, since its directory in `/etc` maps to `/git/apache2`, which includes the same files it did before:

```
sudo systemctl reload apache2
```

If you see no errors, you should be all set. Otherwise, make sure you created the symbolic link properly.

Next, we get to the main attraction. We've copied Apache's files into our repository, but we didn't actually push those changes back to our Git server yet. To set that up, we'll need to associate the files within our `/git/apache2` directory into version control. The reason for this is the files simply being in the `git` repository folder isn't enough for Git to care about them. We have to tell Git to pay attention to individual files. We can add every file within our Git repository for Apache by entering the following command from within that directory:

```
git add
```

This basically tells Git to add everything in the directory to version control. You can actually do the following to add an individual file:

```
git add <filename>
```

In this case, we want to add everything, so we used a period in place of a directory name to add the entire current directory.

If you run the `git status` command from within your Git repository, you should see output indicating that Git has new files that haven't been committed yet. A **Git commit** simply finalizes the changes locally. Basically, it packages up your current changes to prepare them for being copied to the server. To create a commit of all the files we've added so far, `cd` into your `/git/apache2` directory and run the following to stage a new commit:

```
git commit -a -m "My first commit."
```

With this command, the `-a` option tells Git that you want to include anything that's changed in your repository. The `-m` option allows you to attach a message to the commit, which is actually required. If you don't use the `-m` option, it will open your default text editor and allow you to add a comment from there.

Finally, we can `push` our changes back to the Git server:

```
git push origin master
```

By default, the `git` suite of commands utilizes OpenSSH, so our `git push` command should create an SSH connection back to our server and push the files there. You won't be able to inspect the contents of the Git directory on your Git server, because it won't contain the same file structure as your original directory. Whenever you pull a Git repository though, the resulting directory structure will be just as you left it.

From this point forward, if you need to restore a repository onto another server, all you should need to do is perform a Git clone. To clone the repository into your current working directory, execute the following:

```
git clone 192.168.1.101:/git/apache2
```

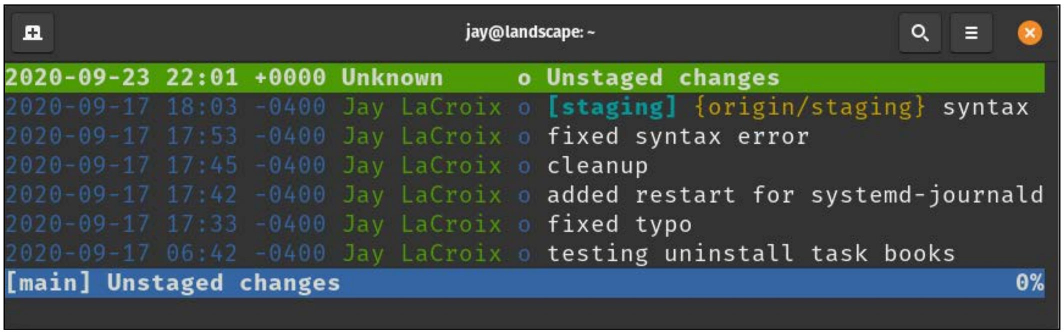
Now, each time you make changes to your configuration files, you can perform a `git commit` and then push the changes up to the server to keep the content safe:

```
git commit -a -m "Updated config files."
git push origin master
```

Now we know how to create a repository, push changes to a server, and pull the changes back down. Finally, we'll need to know how to revert changes should our configuration get changed with non-working files. First, we'll need to locate a known working commit. My favorite method is using the `tig` command. The `tig` package must be installed for this to work, but it's a great utility to have:

```
sudo apt install tig
```

The `tig` command (which is just `git` backward) gives us a semi-graphical interface to browse through our Git commits. To use it, simply execute the `tig` command from within a Git repository. In the following example screenshot, I've executed `tig` from within a Git repository on one of my servers:



```
2020-09-23 22:01 +0000 Unknown      o Unstaged changes
2020-09-17 18:03 -0400 Jay LaCroix o [staging] {origin/staging} syntax
2020-09-17 17:53 -0400 Jay LaCroix o fixed syntax error
2020-09-17 17:45 -0400 Jay LaCroix o cleanup
2020-09-17 17:42 -0400 Jay LaCroix o added restart for systemd-journald
2020-09-17 17:33 -0400 Jay LaCroix o fixed typo
2020-09-17 06:42 -0400 Jay LaCroix o testing uninstall task books
[main] Unstaged changes                                              0%
```

Figure 23.1: An example of the tig command, looking at a repository for the bind9 daemon

While using `tig`, you'll see a list of Git commits, along with their dates and comments that were entered with each. To inspect one, press the up and down arrows to change your selection, then press *Enter* on the one you want to view. You'll see a new window, which will show you the `commit hash` (which is a long string of alphanumeric characters), as well as an overview of which lines were added or removed from the files within the commit. To revert one, you'll first need to find the commit you want to revert to and get its commit hash. The `tig` command is great for finding this information. In most cases, the commit you'll want to revert to is the one *before* the change took place. In my example screenshot, I fixed the syntax issue on 9/17/2020. If I want to restore that file, I should revert to the commit below that. I can get the commit hash by highlighting that entry and pressing *Enter*. It's at the top of the window. Then, I can exit `tig` by pressing *q*, and then revert to that commit:

```
git checkout 356dd6153f187c1918f6e2398aa6d8c20fd26032
```

And just like that, the entire directory tree for the repository instantly changes to exactly what it was before the bad commit took place. I can then restart or reload the daemon for this repository, and it will be back to normal. At this point, you'd want to test the application to make sure that the issue is completely fixed. After some time has passed and you're finished testing, you can make the change permanent. First, we switch back to the most recent commit:

```
git checkout master
```

Then, we permanently switch back to the commit that was found to be working properly:

```
git revert --no-commit 356dd6153f187c1918f6e2398aa6d8c20fd26032
```

Then, we can commit our reverted Git repository and push it back to the server:

```
git commit -a -m "The previous commit broke the application.
Reverting."
git push origin master
```

As you can see, Git is a very useful ally to utilize when managing configuration files on your servers. This benefits disaster recovery, because if a bad change is made that breaks a daemon, you can easily revert the change. If the server were to fail, you can recreate your configuration almost instantly by just cloning the repository again. There's certainly a lot more to Git than what we've gone over in this section, so feel free to pick up a book about it if you wish to take your knowledge to the next level. But in regard to managing your configuration with Git, all you'll need to know is how to place files into version control, update them, and clone them to new servers. Some services you run on a server may not be a good candidate for Git, however. For example, managing an entire MariaDB database via Git would be a nightmare, since there is too much overhead with such a use case, and database entries would likely change too rapidly for Git to keep up. Use your best judgment. If you have some configuration files that are only manipulated every once in a while, they'll be a perfect candidate for Git.

Backups are one of those things that some people don't seem to take seriously until it's too late. Data loss can be a catastrophic event for an organization, so it's imperative that you implement a solid backup plan. In the next section, we'll look at what that entails.

## Implementing a backup plan

Creating a solid backup plan is one of the most important things you'll ever do as a server administrator. Even if you're only using Ubuntu Server at home as a personal file server, backups are critical. During my career, I've seen disks fail many times. I often hear arguments about which hard disk manufacturer beats others in terms of longevity, but I've seen disk failures so often, I don't trust any of them. All disks will fail eventually, it's just a matter of when. And when they do fail, they'll usually fail hard with no easy way to recover data from them. A sound approach to managing data is that any disk or server can fail, and it won't matter, since you'll be able to regenerate your data from other sources, such as a backup or secondary server.

There's no one best backup solution, since it all depends on what kind of data you need to secure, and what software and hardware resources are available to you. For example, if you manage a database that's critical to your company, you should back it up regularly. If you have another server available, set up a replication secondary server so that your primary database isn't a single point of failure. Not everyone has an extra server lying around, so sometimes you have to work with what you have available. This may mean that you'll need to make some compromises, such as creating regular snapshots of your database server's storage volume or regularly dumping a backup of your important databases to an external storage device.

The `rsync` utility is one of the most valuable pieces of software around to server administrators. It allows us to do some really wonderful things. In some cases, it can save us quite a bit of money. For example, online backup solutions are wonderful in the sense that we can use them to store off-site copies of our important files. However, depending on the volume of data, they can be quite expensive. With `rsync`, we can back up our data in much the same way, with not only our current files copied over to a backup target but also differentials. If we have another server to send the backup to, even better.

At one company I managed servers for, they didn't want to subscribe to an online backup solution. To work around that, a server was set up as a backup point for `rsync`. We set up `rsync` to back up to the secondary server, which housed quite a lot of files. Once the initial backup was complete, the secondary server was sent to one of our other offices in another state. From that point forward, we only needed to run `rsync` weekly, to back up everything that had been changed since the last backup. Sending files via `rsync` to the other site over the internet was rather slow, but since the initial backup was already complete before we sent the server there, all we needed to back up each week was differentials. Not only is this an example of how awesome `rsync` is and how we can configure it to do pretty much what paid solutions do but the experience was also a good example of utilizing what you have available to you.

Since we've already gone over `rsync` in *Chapter 12, Sharing and Transferring Files*, I won't repeat too much of that information here. But since we're on the subject of backing up, the `--backup-dir` option is worth mentioning again. This option allows you to copy files that would normally be replaced to another location. As an example, here's the `rsync` command I mentioned in *Chapter 12, Sharing and Transferring Files*:

```
CURDATE=$(date +%m-%d-%Y)
export $CURDATE
sudo rsync -avb --delete --backup-dir=/backup/incremental/$CURDATE /src
/target
```

This command was part of the topic of creating an `rsync` backup script. The first command simply captures today's date and stores it in a variable named `$CURDATE`. In the actual `rsync` command, we refer to this variable. The `-b` option (part of the `-avb` option string) tells `rsync` to make a copy of any file that would normally be replaced. If `rsync` is going to replace a file on the target with a new version, it will move the original file to a new name before overwriting it. The `--backup-dir` option tells `rsync` that when it's about to overwrite a file, to put it somewhere else instead of copying it to a new name. We give the `--backup-dir` option a path, where we want the files that would normally be replaced to be copied to. In this case, the backup directory includes the `$CURDATE` variable, which will be different every day. For example, a backup run on 8/16/2020 would have a backup directory of the following path, if we used the command I gave as an example:

```
/backup/incremental/8-16-2020
```

This essentially allows you to keep differentials. Files on `/src` will still be copied to `/target`, but the directory you identify as `--backup-dir` will contain the original files before they were replaced that day.

On my servers, I use the `--backup-dir` option with `rsync` quite often. I'll typically set up an external backup drive, with the following three folders:

- `Current`
- `Archive`
- `logs`

The `Current` directory always contains a current snapshot of the files on my server. The `Archive` directory on my backup disks is where I point the `--backup-dir` option. Within that directory will be folders named with the dates that the backups were taken. The `logs` directory contains log files from the backup. Basically, I redirect the output of my `rsync` command to a log file within that directory, each log file being named with the same `$CURDATE` variable so I'll also have a backup log for each day the backup runs. I can easily look at any of the logs for which files were modified during that backup, and then traverse the archive folder to find an original copy of a file. I've found this approach to work very well. Of course, this backup is performed with multiple backup disks that are rotated every week, with one always off-site. It's always crucial to keep a backup off-site in case of a situation that could compromise your entire local site.

The `rsync` utility is just one of many you can utilize to create your own backup scheme. The plan you come up with will largely depend on what kind of data you're wanting to protect and what kind of downtime you're willing to endure.

Ideally, we would have an entire warm site with servers that are carbon copies of our production servers, ready to be put into production should any issues arise, but that's also very expensive, and whether you can implement such a routine will depend on your budget. However, Ubuntu has many great utilities available you can use to come up with your own system that works. If nothing else, utilize the power of `rsync` to back up to external disks and/or external sites.

With the concept of cloud computing becoming more and more popular, replacing failed RAID disks is something that you may not ever have to do. But if you do have servers in use that utilize RAID, then it's a good idea to at least understand the procedure in general. In the next section, we'll take a look at that process.

## Replacing failed RAID disks

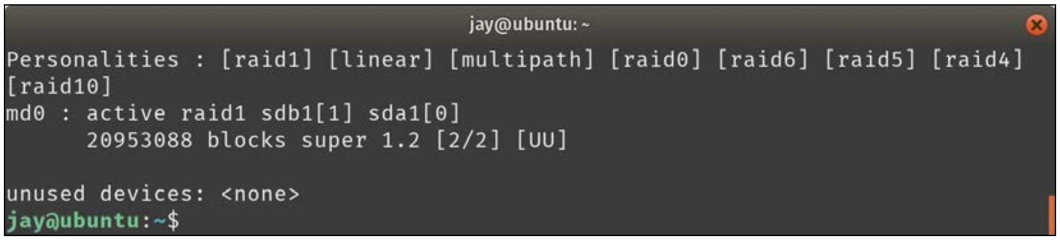
RAID is a very useful technology, as it can help your server survive the crash of a single disk. RAID is not a backup solution, but more of a safety net that will hopefully prevent you from having to reload a server. The idea behind RAID is having redundancy, so that data is mirrored or striped among several disks. With most RAID configurations, you can survive the loss of a single disk, so if a disk fails, you can usually replace it and re-sync and be back to normal. The server itself will continue to work, even if there is a failed disk. However, losing additional disks will likely result in failure right away. When a RAID disk fails, you will need to replace that disk as quick as you can, hopefully before another disk goes too.

> The default live installer for Ubuntu Server doesn't offer a RAID setup option, but the alternate installer does. If you wish to set up Ubuntu Server, check out the *Appendix* at the end of this book.

To check the status of a RAID configuration, you would use the following command:

```
cat /proc/mdstat
```

This will produce output like the following:



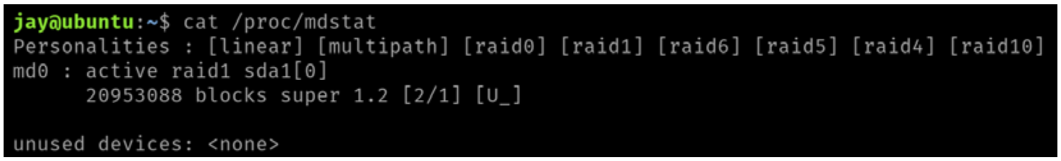
```
Personalities : [raid1] [linear] [multipath] [raid0] [raid6] [raid5] [raid4]
[raid10]
md0 : active raid1 sdb1[1] sda1[0]
      20953088 blocks super 1.2 [2/2] [UU]

unused devices: <none>
jay@ubuntu:~$
```

Figure 23.2: A healthy RAID array

In *Figure 23.2*, we have a RAID 1 array with two disks. We can tell this from the `active raid1` portion of the output. On the next line down, we see this:

```
[UU]
```

Believe it or not, this references a healthy RAID array, which means both disks are online and are working properly. If any one of the U signifiers changes to an underscore, then that means a disk has gone offline and we will need to replace it. Here's a screenshot showing output from that same command on a server with a failed RAID disk:



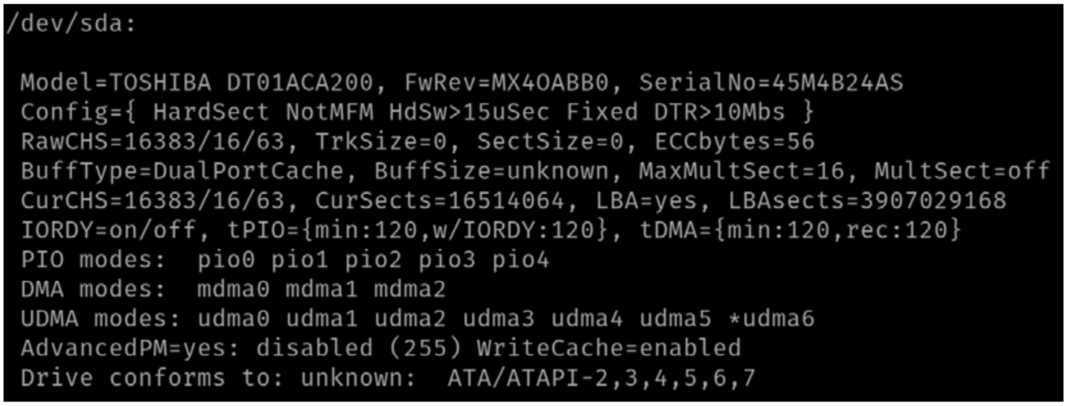
```
jay@ubuntu:~$ cat /proc/mdstat
Personalities : [linear] [multipath] [raid0] [raid1] [raid6] [raid5] [raid4] [raid10]
md0 : active raid1 sda1[0]
      20953088 blocks super 1.2 [2/1] [U_]

unused devices: <none>
```

Figure 23.3: RAID status output with a faulty drive

As you can see from *Figure 23.3*, we have a problem. The `/dev/sda` disk is online, but `/dev/sdb` has gone offline. So, what should we do? First, we would need to make sure we understand which disk is working, and which disk is the one that's faulty. We already know that the disk that's faulty is `/dev/sdb`, but when we open the server's case, we're not going to know *which* disk `/dev/sdb` actually is. If we pull the wrong disk, we could make this problem much worse than it already is. We can use the `hdparm` command to get a little more info from our drive:

```
sudo hdparm -i /dev/sda
```

This will give us info regarding `/dev/sda`, the disk that's currently still functioning properly:



```
/dev/sda:

 Model=TOSHIBA DT01ACA200, FwRev=MX4OABB0, SerialNo=45M4B24AS
 Config={ HardSect NotMFM HdSw>15uSec Fixed DTR>10Mbs }
 RawCHS=16383/16/63, TrkSize=0, SectSize=0, ECCbytes=56
 BuffType=DualPortCache, BuffSize=unknown, MaxMultSect=16, MultSect=off
 CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBAsects=3907029168
 IORDY=on/off, tPIO={min:120,w/IORDY:120}, tDMA={min:120,rec:120}
 PIO modes:  pio0 pio1 pio2 pio3 pio4
 DMA modes:  mdma0 mdma1 mdma2
 UDMA modes: udma0 udma1 udma2 udma3 udma4 udma5 *udma6
 AdvancedPM=yes: disabled (255) WriteCache=enabled
 Drive conforms to: unknown:  ATA/ATAPI-2,3,4,5,6,7
```

Figure 23.4: Output of the hdparm command

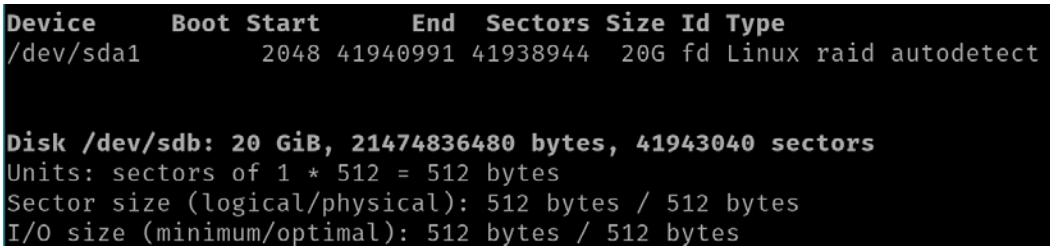
The reason why we're executing this command against a *working* drive is we want to make sure we understand which disk we should NOT remove from the server. Also, the faulty drive may not respond to our attempts to interrogate information from it. Currently, `/dev/sda` is working fine, so we will not want to disconnect the cables attached to that drive at any point. If you have a RAID array with more than two disks, you'll want to execute the `hdparm` command against each. From the output of the `hdparm` command, we can see that `/dev/sda` has a serial number of `45M4B24AS`. When we look inside the case, we can compare the serial number on the drives' labels and make sure we do not remove the drive with this serial number.

Next, assuming we already have the replacement disk on hand, we will want to power down the server. Depending on what the server is used for, we may need to do this after hours, but we typically cannot remove a disk while a server is running. Once it's shut down, we can narrow down which disk `/dev/sdb` is (or whatever drive designation the failed drive has) and replace it. Then, we can power on the server (it will probably take much longer to boot this time; that's to be expected given our current situation).

However, simply adding a replacement disk will not automatically resolve this issue. We need to add the new disk to our RAID array for it to accept it and rebuild the RAID. This is a manual process. The first step in rebuilding the RAID array is finding out which designation our new drive received, so we know which disk we are going to add to the array. After the server boots, execute the following command:

```
sudo fdisk -l
```

You'll see output similar to the following:



```
Device     Boot Start      End  Sectors Size Id Type
/dev/sda1        2048 41940991 41938944  20G fd Linux raid autodetect


Disk /dev/sdb: 20 GiB, 21474836480 bytes, 41943040 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

Figure 23.5: Checking current disks with fdisk

From the output, it should be obvious which disk the new one is. `/dev/sda` is our original disk, and `/dev/sdb` is the one that was just added. To make it more obvious, we can see from the output that `/dev/sda` has a partition, of type `Linux raid autodetect`. `/dev/sdb` doesn't have this.

So now that we know which disk is the new one, we can add it to our RAID array. First, we need to copy over the partition tables from the first disk to the new one. The following command will do that:

```
sudo sfdisk -d /dev/sda | sfdisk sudo /dev/sdb
```

Essentially, we are cloning the partition table from `/dev/sda` (the working drive) to `/dev/sdb` (the one we just replaced). If you run the same `fdisk` command we ran earlier, you should see that they both have partitions of type `Linux raid autodetect` now:

```
sudo fdisk -l
```

Now that the partition table has been taken care of, we can add the replaced disk to our array with the following command:

```
sudo mdadm --manage /dev/md0 --add /dev/sdb1
```

You should see output similar to the following:

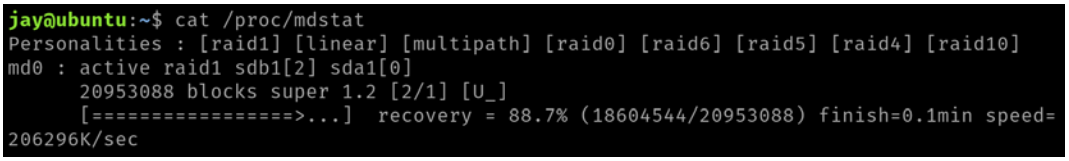
```
mdadm: added /dev/sdb1
```

With this command, we are essentially adding the `/dev/sdb1` disk to a RAID array designated as `/dev/md0`. With the last part, you want to make sure you're executing this command against the correct array designation. If you don't know what that is, you will see it in the output of the `fdisk` command we executed earlier.

Now, we should verify that the RAID array is rebuilding properly. We can check this with the same command we always use to check the RAID status:

```
cat /proc/mdstat
```

This will produce output like the following:



```
jay@ubuntu:~$ cat /proc/mdstat
Personalities : [raid1] [linear] [multipath] [raid0] [raid6] [raid5] [raid4] [raid10]
md0 : active raid1 sdb1[2] sda1[0]
      20953088 blocks super 1.2 [2/1] [U_]
      [================>...]  recovery = 88.7% (18604544/20953088) finish=0.1min speed=
206296K/sec
```

Figure 23.6: Checking the RAID status after replacing a disk

In *Figure 23.6*, you can see that the RAID array is in recovery mode. Recovery mode itself can take quite a while to complete, sometimes even overnight depending on how much data it needs to re-sync. This is why it's very important to replace a RAID disk as soon as possible. Once the recovery is complete, the RAID array is marked healthy and you can rest easy.

A tool that is very valuable when working to recover physical servers is USB recovery media, such as flash drives with a bootable ISO image written to them. In the next section, we'll take a look.

## Utilizing bootable recovery media

The concept of **live media** is a wonderful thing, as we can boot into a completely different working environment from the operating system installed on our device and perform tasks without disrupting installed software on the host system. The desktop version of Ubuntu, for example, offers a complete computing environment we can use in order to not only test hardware and troubleshoot our systems, but also to browse the web just as we would on an installed system. In terms of recovering from disasters, live media becomes a saving grace.

As administrators, we run into one problem after another. This gives us our job security. Computers often seemingly have a mind of their own, failing when least expected (as well as seemingly every holiday). Our servers and desktops can encounter a fault at any time, and live media allows us to separate hardware issues from software issues, by troubleshooting from a known good working environment.

One of my favorites when it comes to live media is the desktop version of Ubuntu. Although geared primarily toward end users who wish to install Ubuntu on a laptop or desktop, as administrators we can use it to boot a machine that normally wouldn't, or even recover data from failed disks. For example, I've used the Ubuntu live media to recover data from both failed Windows and Linux systems, by booting the machine with the live media and utilizing a network connection to move data from the bad machine to a network share. Often, when a computer or server fails to boot, the data on its disk is still accessible. Assuming the disk wasn't encrypted during installation, you should have no problem accessing data on a server or workstation using live media such as the Ubuntu live media.

Sometimes, certain levels of failure require us to use different tools. While Ubuntu's live media is great, it doesn't work for absolutely everything. One situation is a failing disk. Often, you'll be able to recover data using Ubuntu's live media from a failing disk, but if it's too far gone, then the Ubuntu media will have difficulty accessing data from it as well. Tools such as Clonezilla specialize in working with hard disks and may be a better choice.

Live media can totally save the day. The Ubuntu live image in particular is a great boot disk to have available to you, as it gives you a very extensive environment you can use to troubleshoot systems and recover data.

One of the best aspects of using the Ubuntu live image is that you won't have to deal with the underlying operating system and software set at all, you can bypass both by booting into a known working desktop, and then copy any important files from the drive right onto a network share. Another important feature of Ubuntu live media is the memory test option. Quite often, strange failures on a computer can be traced to defective memory. Other than simply letting you install Ubuntu, the live media is a Swiss Army knife of many tools you can use to recover a system from disaster. If nothing else, you can use live media to pinpoint whether a problem is software- or hardware-related. If a problem can only be reproduced in the installed environment but not in a live session, chances are a configuration problem is to blame. If a system also misbehaves in a live environment, it may help you identify a hardware issue. Either way, every good administrator should have live media available to troubleshoot systems and recover data when the need arises.

## Summary

In this chapter, we looked at several ways in which we can prevent and recover from disasters. Having a sound prevention and recovery plan in place is an important key to managing servers efficiently. We need to ensure we have backups of our most important data ready for whenever servers fail, and we should also keep backups of our most important configurations. Ideally, we'll always have a warm site set up with preconfigured servers ready to go in a situation where our primary servers fail, but one of the benefits of open source software is that we have a plethora of tools available to us we can use to create a sound recovery plan. In this chapter, we looked at leveraging `rsync` as a useful utility for creating differential backups, and we also looked into setting up a Git server we can use for configuration management, which is also a crucial aspect of any sound prevention plan. We also talked about the importance of live media in diagnosing issues.

And with this chapter, this book comes to a close. Writing this book has been an extremely joyful experience. I was thrilled to write the first edition when Ubuntu 16.04 was in development, it was a fun project to write the second edition and update it to cover Ubuntu 18.04, and I'm even more thrilled to have had the opportunity to update this body of work for 20.04 in this latest edition. I'd like to thank each and every one of you, my readers, for taking the time to read this book. In addition, I would like to thank all of the viewers of my YouTube channel, *LearnLinux.tv*, because I probably wouldn't have had the opportunity to write this in the first place had it not been for my viewers helping make my channel so popular.

I'd also like to thank Packt Publishing for giving me the opportunity to write a book about one of my favorite technologies. Writing this book was definitely an honor. When I first started with Linux in 2002, I never thought I'd actually be an author, teaching the next generation of Linux administrators the tricks of the trade. I wish each of you the best of luck, and I hope this book is beneficial to you and your career.

For additional content, be sure to check out `https://learnlinux.tv` for even more content. I have quite a few training videos available there, free of charge.

## Further reading

- *Pro Git* by Scott Chacon and Ben Straub: `https://git-scm.com/book/en/v2`
- *Introduction to RAID terminology and concepts*: `https://www.digitalocean.com/community/tutorials/an-introduction-to-raid-terminology-and-concepts`

# Another Book You May Enjoy

# Index