

Mastering Ubuntu Server

Second Edition

Master the art of deploying, configuring, managing,
and troubleshooting Ubuntu Server 18.04



By Jay LaCroix

Packt

www.packt.com

Mastering Ubuntu Server

Second Edition

Master the art of deploying, configuring, managing, and troubleshooting Ubuntu Server
18.04

Jay LaCroix

Packt

BIRMINGHAM - MUMBAI

Mastering Ubuntu Server Second Edition

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Gebin George

Acquisition Editor: Heramb Bhavsar

Content Development Editor: Nithin George Varghese

Technical Editor: Vishal Kamal Mewada

Copy Editor: Safis Editing

Project Coordinator: Virginia Dias

Proofreader: Safis Editing

Indexer: Rekha Nair

Graphics: Tom Scaria

Production Coordinator: Nilesh Mohite

First published: July 2016

Second edition: May 2018

Production reference: 1240518

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78899-756-0

www.packtpub.com

To Johnny and Alan, I love you both and am proud of you each and every day.



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Jay LaCroix is a technologist and open source enthusiast, specializing in Linux. He currently works as a senior solutions architect and holds a master's degree in information systems technology management from Capella University. In addition, Jay also has a YouTube channel, available at LearnLinux.tv, where he posts instructional tutorial videos. He has also written *Linux Mint Essentials* and *Mastering Linux Network Administration*, also published by Packt Publishing.

I would like to thank Randy Schapel of Mott Community College; his Linux class so many years ago is what started my career (and obsession). I would also like to thank my family and friends for their support. Also, thank you to all of my YouTube subscribers and viewers, as passing along my knowledge to you has been a wonderful experience.

About the reviewers

David Diperna is a cloud engineer who graduated from Oakland University with degrees in computer science and psychology. He has two AWS certifications, and he works on multiple projects building scalable infrastructure and custom solutions for different apps running on Linux.

His work requires a wide array of skill sets, but he specializes in metric and logging systems, such as Graphite and Elastic Stack, for application performance monitoring and analysis.

Matthew Huber has been working in the tech industry for 20 years, specializing in Unix and Linux operating systems. He has helped defined system standards and led migration teams, and he was a Solaris subject matter expert for big three automakers. He is currently a cloud engineer, designing and implementing solutions in AWS.

He was on the board of directors of i3Detroit and led a team in designing and competing in electric vehicles in the Power Racing Series.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Title Page	
Copyright and Credits	
Mastering Ubuntu Server Second Edition	
Dedication	
Packt Upsell	
Why subscribe?	
PacktPub.com	
Contributors	
About the author	
About the reviewers	
Packt is searching for authors like you	
Preface	
Who this book is for	
What this book covers	
To get the most out of this book	
Conventions used	
Get in touch	
Reviews	
1. Deploying Ubuntu Server	
Technical requirements	
Setting up our lab	
Determining your server's role	
Deciding between 32- and 64-bit installations	
Obtaining Ubuntu Server installation media	
Creating a bootable Ubuntu Server flash drive	
Planning the partitioning layout	
Installing Ubuntu Server	
Installing Ubuntu Server on Raspberry Pi 3	
Summary	
Questions	
Further reading	
2. Managing Users	
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
Summary
Questions
Further reading

3. Managing Storage Volumes

Understanding the Linux filesystem
Using symbolic and hard links
Viewing disk usage
Adding additional storage volumes
Partitioning and formatting volumes
Mounting and unmounting volumes
Understanding the /etc/fstab file
Managing swap
Utilizing LVM volumes
Understanding RAID
Summary
Questions
Further reading

4. Connecting to Networks

Setting the hostname
Managing network interfaces
Assigning static IP addresses
Understanding NetworkManager
Understanding Linux name resolution
Getting started with OpenSSH
Getting started with SSH key management
Simplifying SSH connections with a config file
Summary
Questions
Further reading

5. Managing Software Packages

Understanding Linux package management
Taking advantage of hardware enablement updates
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](#)

[Making use of Aptitude](#)

[Summary](#)

[Questions](#)

6. Controlling and Monitoring Processes

[Showing running processes with the ps command](#)

[Managing jobs](#)

[Dealing with misbehaving processes](#)

[Utilizing htop](#)

[Managing system processes](#)

[Monitoring memory usage](#)

[Scheduling tasks with cron](#)

[Understanding load average](#)

[Summary](#)

[Questions](#)

[Further reading](#)

7. Setting Up Network Services

[Planning your IP address scheme](#)

[Serving IP addresses with isc-dhcp-server](#)

[Setting up DNS with bind](#)

[Creating a secondary \(slave\) DNS server](#)

[Setting up an internet gateway](#)

[Keeping your clock in sync with NTP](#)

[Summary](#)

[Questions](#)

[Further reading](#)

8. Sharing and Transferring Files

[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](#)

[Summary](#)

[Questions](#)

[Further reading](#)

9. Managing Databases

[Preparations for setting up a database server](#)

[Installing MariaDB](#)

[Understanding the MariaDB configuration files](#)

[Managing MariaDB databases](#)
[Setting up a slave database server](#)
[Summary](#)
[Questions](#)

10. [Serving Web Content](#)

[Installing and configuring Apache](#)
[Installing additional Apache modules](#)
[Securing Apache with SSL](#)
[Installing and configuring NGINX](#)
[Setting up failover with keepalived](#)
[Setting up and configuring Nextcloud](#)
[Summary](#)
[Questions](#)
[Further reading](#)

11. [Learning Advanced Shell Techniques](#)

[Understanding the Linux shell](#)
[Understanding Bash history](#)
[Learning some useful command-line tricks](#)
[Redirecting output](#)
[Understanding variables](#)
[Writing simple scripts](#)
[Putting it all together: Writing an rsync backup script](#)
[Summary](#)
[Questions](#)
[Further reading](#)

12. [Virtualization](#)

[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](#)
[Summary](#)
[Questions](#)
[Further reading](#)

13. [Running Containers](#)

[What is containerization?](#)
[Understanding the differences between Docker and LXD](#)
[Installing Docker](#)
[Managing Docker containers](#)
[Automating Docker image creation with Dockerfiles](#)

[Managing LXD containers](#)

[Summary](#)

[Questions](#)

[Further reading](#)

14. Automating Server Configuration with Ansible

[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](#)

[Summary](#)

[Questions](#)

[Further reading](#)

15. Securing Your Server

[Lowering your attack surface](#)

[Understanding and responding to CVEs](#)

[Installing security updates](#)

[Automatically installing patches with the Canonical Livepatch service](#)

[Monitoring Ubuntu servers with Canonical's 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](#)

[Summary](#)

[Questions](#)

[Further reading](#)

16. Troubleshooting Ubuntu Servers

[Evaluating the problem space](#)

[Conducting a root cause analysis](#)

[Viewing system logs](#)

[Tracing network issues](#)

[Troubleshooting resource issues](#)

[Diagnosing defective RAM](#)

[Summary](#)

[Questions](#)

17. Preventing and Recovering from Disasters

[Preventing disasters](#)

[Utilizing Git for configuration management](#)

[Implementing a backup plan](#)

[Replacing failed RAID disks](#)

[Utilizing bootable recovery media](#)

[Summary](#)

[Questions](#)

[Further Reading](#)

[Using the Alternate Installer](#)

[Obtaining the Alternate Installer](#)

[Installing via the Alternate Installer](#)

[Setting up software RAID](#)

[Summary](#)

[Assessments](#)

[Chapter 1; Deploying Ubuntu Server](#)

[Chapter 2; Managing Users](#)

[Chapter 3; Managing Storage Volumes](#)

[Chapter 4; Connecting to Networks](#)

[Chapter 5; Managing Software Packages](#)

[Chapter 6; Controlling and Monitoring Processes](#)

[Chapter 7; Setting Up Network Services](#)

[Chapter 8; Accessing and sharing files](#)

[Chapter 9; Sharing and Transferring Files](#)

[Chapter 10; Serving Web Content](#)

[Chapter 11; Learning Advanced Shell Techniques](#)

[Chapter 12; Virtualization](#)

[Chapter 13; Running Containers](#)

[Chapter 14; Automating Server Configuration with Ansible](#)

[Chapter 15; Securing Your Server](#)

[Chapter 16; Troubleshooting Ubuntu Servers](#)

[Chapter 17; Preventing and Recovering from Disasters](#)

[Other Books You May Enjoy](#)

[Leave a review - let other readers know what you think](#)

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 a good company—according to W3Techs, Ubuntu is the most widely deployed distribution on the web with regard to Linux. With the release of Ubuntu 18.04, 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, and sharing data with other users.

We'll start our journey right in the first chapter, where we'll walk through the installation of Ubuntu Server 18.04, which will serve as the 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, covers user management in full. Topics here will include creating and removing users, password policies, the sudo command, as well as group management and switching from one user to another.

[Chapter 3](#), 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 and creating links.

[Chapter 4](#), 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, as well as name resolution.

[Chapter 5](#), 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 6](#), Controlling and Monitoring 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, managing jobs, and understanding the load average.

[Chapter 7](#), 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 8](#), Sharing and Transferring Files, is all about sharing files with others. Concepts will include the set up of Samba and NFS network shares, and we will even go over transferring files manually with rsync and scp.

[Chapter 9](#), 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 slave database server.

[Chapter 10](#), 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 11](#), Learning Advanced Shell Techniques, goes over additional tips, tricks, and techniques to enhance the reader's usage of command lines. Topics here include managing output, setting up aliases, investigating Bash history, and more.

[Chapter 12](#), 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 13](#), Running Containers, discusses the subject of containers and show the reader how to manage containers in both Docker and LXD.

[Chapter 14](#), 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 to automate common administrative tasks, and also how to use ansible-pull.

[Chapter 15](#), Securing Your Server, takes a look at various things the reader can do to strengthen security on Ubuntu servers. Topics will include concepts such as lowering the attack surface, securing OpenSSH, setting up a firewall, and more.

[Chapter 16](#), Troubleshooting Ubuntu Servers, consists of topics relating to things we can do when our deployments don't go exactly according to plan. The reader will also investigate the problem space, view system logs, and trace network issues.

[Chapter 17](#), Preventing and Recovering from 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.

[Appendix](#), Using the Alternate Installer, shows the reader how to utilize an alternative installer for Ubuntu that can be used to set up more advanced installations, such as Ubuntu on RAID.

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, not 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.

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: "I added an IP address of `192.168.0.101` with a CIDR mask of `/24`."

A block of code is set as follows:

```
# This file describes the network interfaces available on your system
# For more information, see netplan(5).
network:
  version: 2
  renderer: networkd
  ethernets:
    enp0s3:
      dhcp4: yes
```

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

```
# This file describes the network interfaces available on your system
# For more information, see netplan(5).
network:
  version: 2
  renderer: networkd
  ethernets:
    enp0s3:
      dhcp4: no
      addresses: [192.168.0.101/24]
      gateway4: 192.168.1.1
      nameservers:
        addresses: [192.168.1.1,8.8.8.8]
```

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

```
sudo ip link set enp0s3 down
sudo ip link set enp0s3 up
```

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: "At this point, you'll click Select image, 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."



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 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 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 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.

Deploying Ubuntu Server

Ubuntu Server is an extremely powerful distribution 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. In this chapter in particular, I'll guide you through the process of deploying Ubuntu Server on to your system. We'll start off with some discussion on 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. I'll even show you the process of installing Ubuntu Server on a Raspberry Pi 3 for good measure.

In particular, we will cover:

- Setting up our lab
- Determining your server's role
- Deciding between 32- and 64-bit installations
- Obtaining Ubuntu Server installation media
- Creating a bootable Ubuntu Server flash drive
- Planning the partition layout
- Installing Ubuntu Server
- Installing Ubuntu Server on Raspberry Pi 3

Technical requirements

For this chapter (and all others) we will need either a physical computer or virtual machine with the following specs:

- One CPU core
- 512 MB RAM
- 10 GB hard disk (16 or more recommended)

Don't worry about the specifics for now; we will discuss these requirements further in this chapter.

Setting up our lab

The first thing for us to do is to set up our environment. For the purposes of this book, it really doesn't matter what kind of hardware you use, if you even use physical hardware at all (a virtual machine or cloud instance will suffice). If you have physical server hardware available to you, then by all means use it (though not everyone has the money or even the available space to set up a complete lab). Nowadays, processing power and memory is cheaper than it's ever been, and the typical home PC is capable of running several virtual machines. Even better, software such as VirtualBox is free, so even if you have no budget at all, there are definitely options available to you. Perhaps you already have a server or two in your rack at work just waiting to be deployed; in which case, this will be even more exciting and fun. In the case of VirtualBox, it can be downloaded from here: <https://www.virtualbox.org>.

While I always prefer physical hardware, virtual machines have one major advantage. Not only are they easy to create and destroy at will, they are perfect for testing new or proposed solutions because you can easily take a snapshot of a VM and restore it if an experiment blows up in your face. In my case, it's common for me to create a virtual machine of a Linux distribution, fully update it, and then take a snapshot before making any changes. That way, I always have a base version of the OS to fall back on if I muck it up. Testing software rollouts on virtual machines before graduating a solution to production is a good habit to get into anyway.

This book makes no assumptions regarding your environment, because the hardware you have access to differs from person to person. While the installation procedure for Ubuntu Server differs based on hardware (for example, perhaps your server isn't capable of booting from USB install media, forcing you to use a DVD), all other concepts contained throughout the book will be the same regardless.

With that said, once we run through the install procedure, feel free to install Ubuntu Server on as many devices or VMs as you can. Later on in our journey, we'll go through several examples of networking, including copying files from one server to another. If you have more than one installation, these exercises will be easier. You can set up multiple virtual machines, physical servers, or any combination of the two.

If you don't have access to a physical server, and your PC isn't powerful enough to handle virtual machines very well, consider setting up a **Virtual Private Server (VPS)**

from providers such as **Linode** and **DigitalOcean**, which both have Ubuntu Server available as an option for a small monthly fee. See the following URLs for more information on two popular VPS providers:

- **DigitalOcean:** <https://www.digitalocean.com>
- **Linode:** <https://www.linode.com>

Cloud providers are great if you wish to get a server up and running quickly or if your goal is to set up a server to be available in the cloud. With that said, all you need to do is have Ubuntu Server installed on something (basically anything at all) and you're ready to follow along.

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 concepts 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, also known as its **role**. This role could be that of a database server, web 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 importance. 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 colleague's 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 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 miscreant 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 definitely keep your server room locked!

At this point in the book, I'm definitely not asking you to create a detailed flow chart 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.

Deciding between 32- and 64-bit installations

Before we grab our installation media and get started, I thought I would write a bit about the diminishing availability of 32-bit installation media in the Linux world. When the first edition of this book was published, I wrote about the choice between 32- and 64-bit installations. Nowadays, whether to use 64-bit installation media is no longer much of an option, with various distributions of Linux dropping support for 32-bit downloads. In the case of Ubuntu Server 18.04, you are now only able to download 64-bit installation images as 32-bit downloads of Ubuntu Server have been removed.

This may seem like a polarizing decision on Canonical's part, but it's really not as bad as it may seem. Not being able to download 32-bit installation images from Canonical doesn't mean that you can't run 32-bit software; it just means that your overall system will be running a 64-bit kernel. This is great, considering you'll benefit fully by being able to more efficiently utilize the RAM in your server.

In addition, 64-bit capable processors have been on the market for well over a decade. Even if you think your hardware may be too old to utilize 64-bit software, it will more than likely support it just fine. Consider this—several versions of the Pentium 4 processor support 64-bit software, and that processor has become ancient history in computer years. One scenario that may suffer due to the decision to decommission 32-bit media is installation on **netbooks**, but most people don't run server applications from such a device, so that doesn't affect us at all in terms of this book.

All in all, the decision to migrate away from 32-bit is a good move in the name of progress, and it shouldn't impact us at all when setting up new servers. However, I mentioned it here simply because I wanted to make you aware of it. If you are downloading a modern version of Ubuntu Server today, you're downloading the 64-bit version (and you probably won't notice a difference).

Obtaining Ubuntu Server installation media

It's time to get started! At this point, we'll need to get our hands on Ubuntu Server and then create bootable installation media to install it. How you do this largely depends on your hardware. Does your server have an optical drive? Is it able to boot from USB? Refer to the documentation for your server to find out. In most cases, it's a very good idea to create both a bootable DVD and USB media of Ubuntu Server while you're at it. That way, regardless of how your server boots, you'll have what you need.



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 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 and only ship with an optical drive if you ask for one nicely while placing your order. If a server does have an optical drive, it typically will 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 18.04 LTS by clicking on the Download button. There may be other versions of Ubuntu Server listed on this page, such as 18.10 and 19.04, for example. However, we're only interested in 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). Non-LTS releases are useful for testing future software versions, but when in doubt, stick with LTS. Once the download

is completed, we'll end up with an ISO image we can use to create our bootable installation media.



The ISO image we just downloaded is known as the live installer. This version is the default, and is preferred for most use cases. However, if you are performing a more advanced installation (such as installing Ubuntu with RAID) you may need the alternative installer instead. If you do need this version, you can get it by clicking on the link for alternative downloads page, and then on the next page you'll see an option for the Alternative Ubuntu Server installer. This special installer is covered in the appendix at the back of the book. Otherwise, we'll continue on with the standard installer.

If you're setting up a virtual machine, 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 case, 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. Going over the process of booting an ISO image on a virtual machine differs from hypervisor to hypervisor, 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 virtual machine, and then starting it up.

As I mentioned before, I recommend creating both a bootable USB and bootable DVD. This is due to the fact that you'll probably run into situations where you have a server that doesn't boot from USB, or perhaps your server doesn't have an optical drive and the USB flash drive is your only option. In addition, the Ubuntu Server boot media also makes great recovery media if for some reason you need to rescue a server. 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 disc or some similar verbiage. This is true of Windows, as well as most graphical desktop environments of Linux when a disc burning application installed. If in doubt, **Brasero** is a good disc-burning utility to download for Linux, and other operating systems generally have this option built-in nowadays.

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 disc is not what you want, so make sure to look for the option to create media from an ISO image or your disc 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.

Creating a bootable Ubuntu Server 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. In the first edition, I outlined the process on all three major platforms. Thankfully, a much simpler method has come about since the publication of the first edition. 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 OS 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 out everything. Etcher doesn't let you do that.

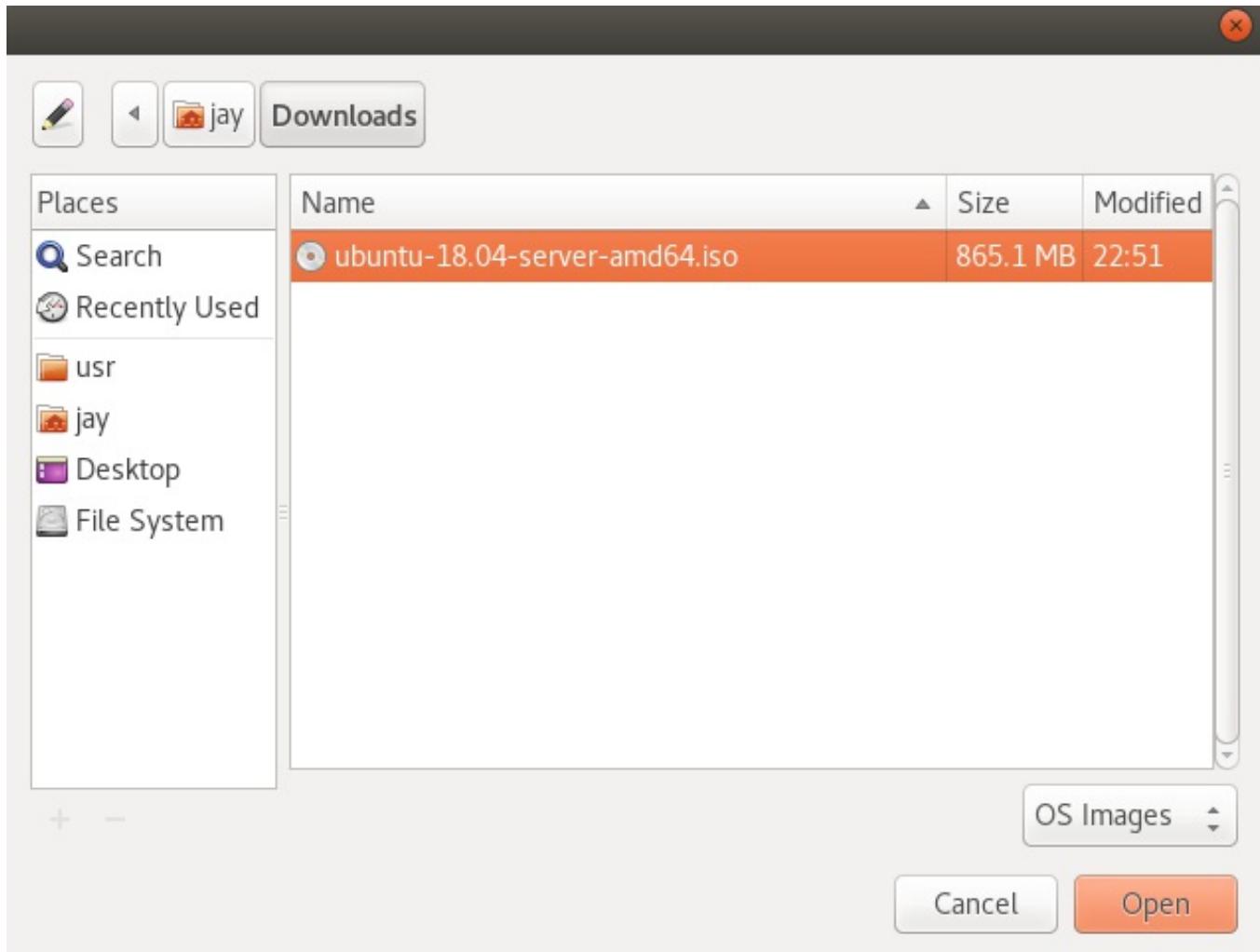
 Before continuing, you'll need a USB flash drive that is either empty, or one you don't mind wiping. This process will completely erase it, so make sure the device doesn't have information on it that you'd rather not lose. The flash drive should be at least 1 GB, preferably 2 GB or larger. Considering 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://etcher.io>, 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:



Utilizing Etcher to create a bootable flash drive

At this point, you'll click Select image, 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:



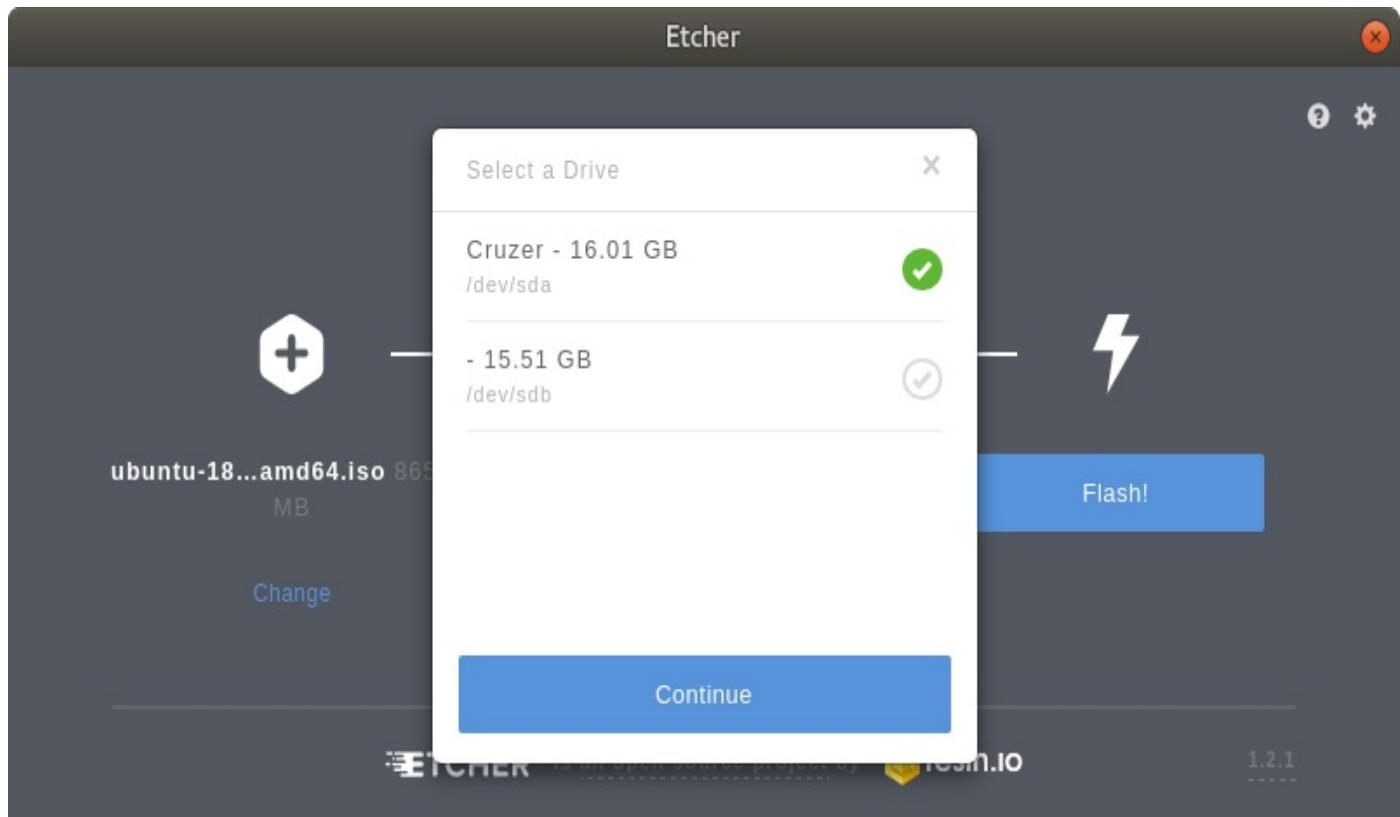
Selecting an ISO image with Etcher

If your flash drive is already inserted into the computer, Etcher should automatically detect it:



Etcher with a selected ISO image and flash drive

In the event 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:



Selecting a different flash drive with Etcher

Finally, you can click Flash! 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:



Selecting a different flash drive with Etcher

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

One of the many joys of utilizing Linux for server platforms is how flexible the partitioning is. **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.

Admittedly, we're probably 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're 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 backup partitions that have data you care about (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 backup 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 administrator when it comes to storage. While helpful if you're trying to figure out why something crashed, logs can fill up a hard disk if you're not careful. In my experience, I've seen servers 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.

While there are certainly better ways of handling excessive logging (log rotating, disk quotas, and so on), a separate partition also would have helped. 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. A full log partition will certainly cause issues, but it wouldn't have been able to affect the entire server. As an administrator, it's up to you to weigh the pros and the cons 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—and 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, figuring out how to break it in epic ways, and then fixing it up.

Here are some basic tips regarding partitioning:

- At minimum, a partition for the root filesystem (designated by a forward slash) is required.
- The `/var` directory contains most log files, and is a ripe candidate for separation.
- The `/home` directory stores all user files. Separating this into a separate partition can be beneficial.
- If you've used Linux before, you may be familiar with the concept of a swap

partition. This is no longer necessary—a swap file will be created automatically in newer Ubuntu releases.

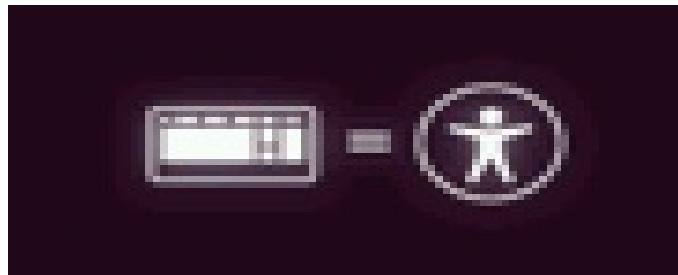
When we perform our installation in the next section, we'll choose the defaults for the partitioning scheme to get you started quickly. However, I recommend you come back to the installation process at some point in the future and experiment with it. You may come up with some clever ways to split up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

Installing Ubuntu Server

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

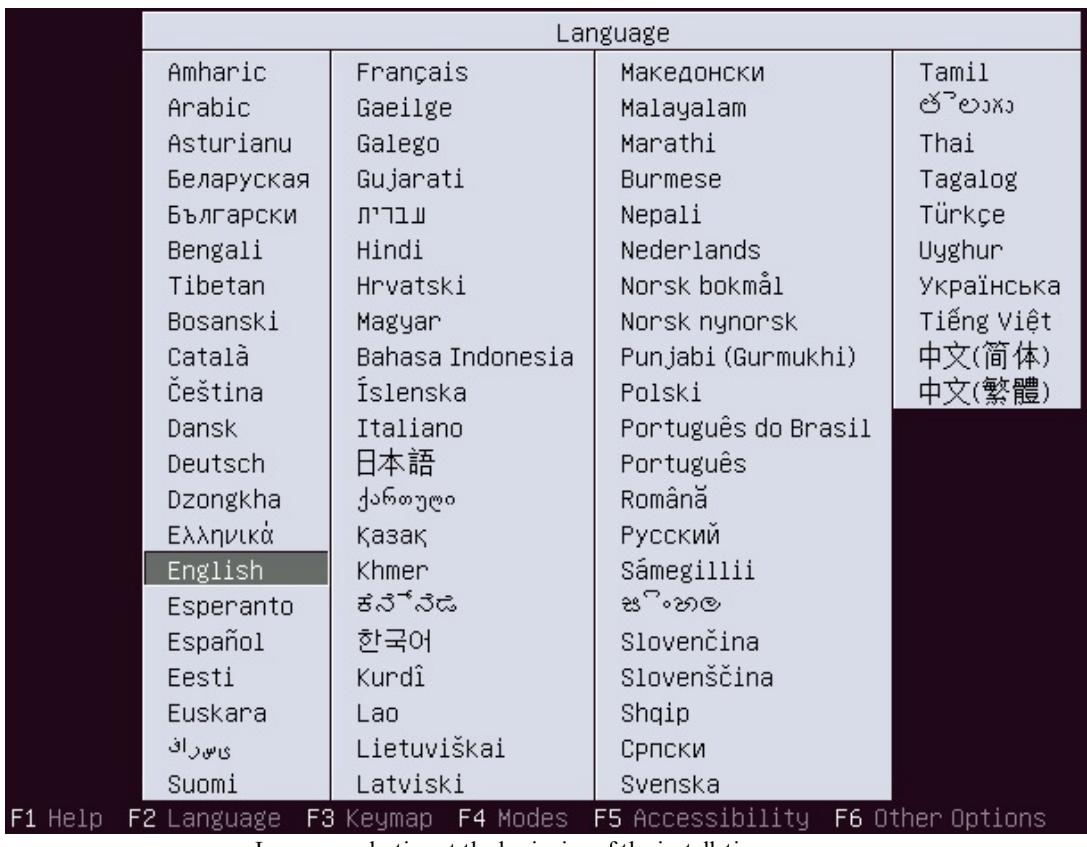
To get started, all you should need to do is insert the media into your server or device and follow the onscreen instructions to open the boot menu. The key you press at the beginning of the POST process differs from one machine to another, but it's quite often F10, F11, or F12. Refer to your documentation if you are unsure, though most computers and servers tell you which key to press at the beginning. You'll probably 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):



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, this will be bypassed and the default options (such as the English language) will be chosen automatically. If you do press Enter, you'll see the following screenshot:



If your language is anything other than English, you'll be able to select that here. You can also press F6 to view additional 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:



Main menu of the Ubuntu installer

First, this menu also offers you an option to Check disc for defects. Although I'm sure you've done well in creating your media, all it takes is for a small corruption in the download and you would end up with invalid media. This is extremely rare—but it does happen from time to time. If you have the extra time, it may make sense to verify your media.



If boot media created from your PC is constantly corrupted or invalid, try creating the media from another machine and test your PC's memory. You can test memory right from the Ubuntu Server installation media.

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. 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 ever been tested at all. In fact, I make it a process to test the RAM of all my servers once or twice a year, so this media 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.

From this point forward, we will progress through the various screens to customize our

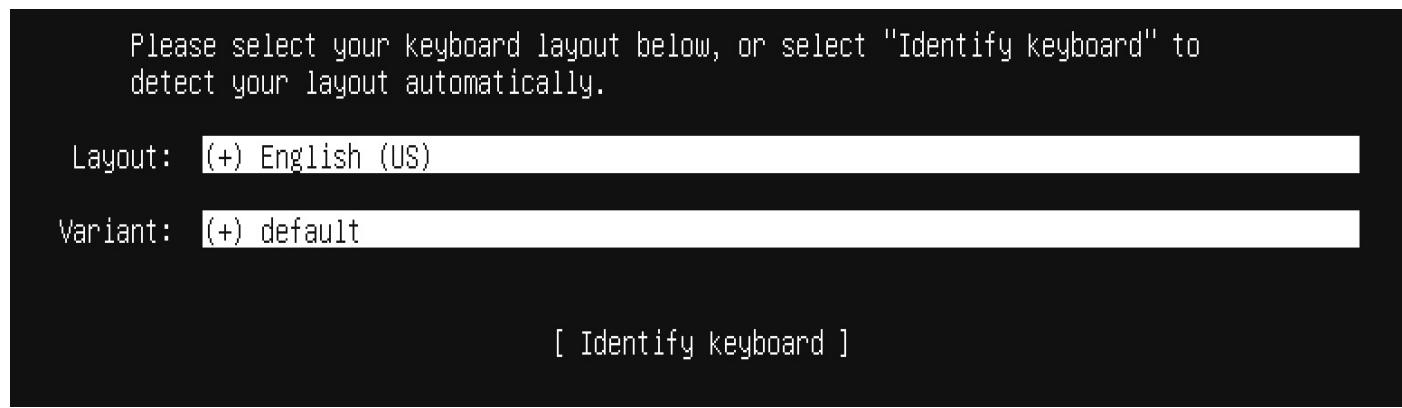
installation. I'll guide you along the way.

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:



Language selection screen

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 down key to highlight Done and press Enter when you've finished:



Choosing a keyboard layout

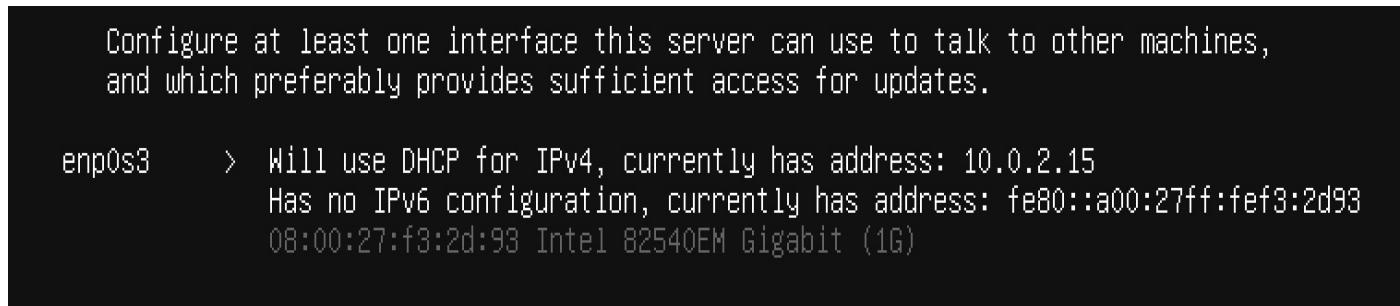
At the next screen, you will have the option to install Ubuntu Server, or a MAAS region or rack controller. **MAAS**, or **Metal as a Service**, is beyond the scope of this book, but it offers some very interesting features, such as additional options for scaling and provisioning your servers. After you become proficient with Ubuntu Server, it may be worth learning later. I have included a link to the documentation for MAAS at the end of this chapter, should you desire to learn more about it. For now, choose the first option,

Install Ubuntu:



Choosing to install Ubuntu or a MAAS controller

Next, the installer will automatically try to configure your network connection via DHCP. This is often correct, so you shouldn't need to do anything here unless you have more than one network card and you'd like to use one other than the default. In my case, it defaulted to device `enp0s3` (which is correct) but the interface name will be different depending on your hardware. If you do need to change this, you can select the defaulted interface name, press Enter, and select a different one. When you're finished, arrow down to done and press Enter:



Network configuration step of the Ubuntu Server installer

The next screen will give you a chance to configure your server's disk. At this screen, if you want to implement a custom partitioning scheme, you can do that here. The option Manual 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:

The installer can guide you through partitioning a disk or, if you prefer, you can do it manually. If you choose guided partitioning you will still have a chance to review and modify the results.

```
[ Use An Entire Disk ]
[ Manual                ]
[ Back                  ]
```

Disk setup screen of the Ubuntu Server installer

After selecting Use An Entire Disk, the next screen will allow you to choose a disk on which to install Ubuntu Server. In my case, I only have one disk, and you probably do as well. But if you have more than one, select the drive to be used as the main drive for the distribution. We can always configure any other drives you may have later. In fact, we will take a look at storage volumes in [Chapter 3](#), Managing Storage Volumes. Simply press Enter after highlighting the appropriate disk:

```
Choose the disk to install to:
```

```
VBOX_HARDDISK_VBeb947136-e95508b9      15.998G >
```

```
[ Cancel    ]
```

Choosing a disk to install Ubuntu Server on

Next, the Ubuntu installer will give you an overview of the partitioning it will perform. The defaults are fine, so simply press Enter. You'll be asked to confirm one last time; when you are, arrow down to continue and press Enter:

FILE SYSTEM SUMMARY

MOUNT POINT	SIZE	TYPE	DEVICE TYPE
/	15.996G	ext4	partition of local disk

AVAILABLE DEVICES

DEVICE	SIZE	TYPE
VBOX_HARDDISK_VBeb947136-e95508b9	15.998G	local disk
partition 1, unformatted	2.000M (0%)	
partition 2, ext4, /	15.996G (99%)	

```
Edit Partitions >
```

Confirming the default partitioning scheme

At the next screen, you'll fill out information relative to the default user account. Fill out

your information on this screen, as I have in the sample 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 (or ssh identity) you will use to log in to the system.

Your name: Jay LaCroix

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

Pick a username: jay

Choose a password: *****

Confirm your password: *****

Import SSH identity: (+) No
You can import your SSH keys from Github, Launchpad or Ubuntu One.

Confirming the default partitioning scheme



While you're in the process of typing in your user info, Ubuntu Server will actually start installing in the background. Talk about being efficient!

At this point, the installation will continue on, and may even be well on its way before you're finished filling out your user information. You'll see a screen similar to the following when everything is finished. Press Enter to reboot the server into the new installation:

```
----- Finished install! -----
finish: cmd-install/stage-curthooks/builtin: SUCCESS: running 'curtin
curthooks'
builtin took 11.409 seconds
stage_curthooks took 11.409 seconds
finish: cmd-install/stage-curthooks: SUCCESS: configuring installed system
start: cmd-install/stage-hook: finalizing installation
start: cmd-install/stage-hook/builtin: running 'curtin hook'
start: cmd-install/stage-hook/builtin/cmd-hook: curtin command hook
Finalizing /target
finish: cmd-install/stage-hook/builtin/cmd-hook: SUCCESS: curtin command
hook
finish: cmd-install/stage-hook/builtin: SUCCESS: running 'curtin hook'
builtin took 0.538 seconds
stage_hook took 0.538 seconds
finish: cmd-install/stage-hook: SUCCESS: finalizing installation
start: cmd-install/stage-late: executing late commands
stage_late took 0.000 seconds
finish: cmd-install/stage-late: SUCCESS: executing late commands
curtin: Installation finished.
Copying curtin install log from /var/log/curtin/install.log to
target//var/log/installer/curtin-install.log
Skipping unmount: config disabled target unmounting
finish: cmd-install: SUCCESS: curtin command install
```

[Reboot Now]

Installation is now complete

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

Installing Ubuntu Server on Raspberry Pi 3

The Raspberry Pi 3 has become quite a valuable asset in the industry. These tiny computers, now with four cores and 1 GB of RAM, are extremely useful in running one-off tasks and even offering a great place to test code without disrupting existing infrastructure. In my lab, I have several Pis on my network, each one responsible for performing specific tasks or functions. It's a very useful platform.

Thankfully, Ubuntu Server is quite easy to install on a Raspberry Pi. In fact, it's installed in much the same way as **Raspbian**, which is the distribution most commonly used on the Pi. Ubuntu Server is available for Raspberry Pi models 2 and 3. All you'll need to do is download the SD card image and then write it directly to your microSD card (your SD card needs to be at least 4 GB). Once you've done that, you can boot the Pi and log in. The default username and password is `ubuntu` for both.

To get started, you'll first download the SD card image from here: <https://wiki.ubuntu.com/ARM/RaspberryPi>.

Then, check that it is compatible with your model of Raspberry Pi. The image will be in a compressed `.xz` format, which you'll need to extract with your operating system's decompressing utility (in Linux you can use the `unxz` command, which you get from the `xz-utils` package).

Next, we do the actual writing of the SD card. For this, we return to our old friend Etcher, which we used earlier in this chapter to master our USB media. Another feature of Etcher is that it allows us to master SD cards for the Raspberry Pi as well. Go ahead and open Etcher, and select the Ubuntu Server Pi image in much the same way as you selected the Ubuntu Server ISO file earlier. Next, make sure your SD card is inserted, and click Flash! and you should be good to go as soon as the process concludes. Once the process finishes, simply insert the SD card into your Pi and power it on. That's all there is to it!

Summary

In this chapter, we covered the installation process in great detail. As with most Linux distributions, Ubuntu Server is scalable and able to be installed on a variety of server types. Physical servers, virtual machines, and even the Raspberry Pi, have a version of Ubuntu Server available. The process of installation was covered step by step, and you should now have an Ubuntu Server of your own to configure as you wish. Also in this chapter, we covered partitioning, creating boot media, determining the role of your server, as well as some best practices.

In the next chapter, I'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. I'll also cover permissions so that you can determine what your users are allowed to do on your server. See you there!

Questions

In our first chapter, we covered the installation process that will serve as the foundation for the rest of the book. However, there were still some useful concepts that we'll want to remember. Take a look at the following questions and test your knowledge:

1. Determining your server's _____ will help you plan for its use on your network.
2. Ubuntu Server installation media is created from an _____ file, which you can download from the Ubuntu website.
3. Name at least one provider for VPS solutions (if you can name more than one, even better).
4. What type of encryption is used when encrypting your Ubuntu installation?
5. What is an LTS release of Ubuntu? Why is it important?
6. Which tool is useful for creating bootable USB flash drives and optical media for installation?
7. At minimum, your Ubuntu Server installation requires a partition for _____ .

Further reading

At the end of some chapters, I will list resources you can use to explore additional concepts relative to the content of the chapter. While not required, you should definitely explore them as these resources will go over additional content that didn't fit the scope of the chapter but are still useful for expanding your knowledge even further:

- **Ubuntu installation documentation:** <https://help.ubuntu.com/community/Installation>
- **Etcher user documentation:** <https://github.com/resin-io/etcher/blob/master/docs/USER-DOCUMENTATION.md>
- **MAAS documentation:** <https://docs.maas.io/2.1/en/>

Managing Users

As an administrator of Ubuntu-based 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 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

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 as an administrator of the system. So, at this point, we should have 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.

In regard to `root`, 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, and I do mean anything. Want to use `root` to create files and directories virtually anywhere on the filesystem? No problem. Want to use `root` to install software? Again, not a problem. The `root` account can even be used to destroy your entire installation with one typo or ill-conceived command. Even if you instruct `root` to delete all 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 instructs, or at least highly recommends, that you create a standard user during the installation process. It's generally recommended that you not use the `root` account unless you absolutely have to. It's common in the Linux community for an administrator to have his or her own account and then switch to `root` whenever a task comes up that requires `root` privileges to complete. It's less likely to destroy your server with an accidental typo or bad command while you're not logged in as `root`. Don't get me wrong, some arrogant administrators will strictly use `root` at all times without any issue, but again, it's recommended to use `root` only when you have to. (And besides, when it comes to arrogant administrators, don't be that person.)

Most distributions ask you to create a `root` password during installation in order to protect that account. Ubuntu Server is a bit different in this regard, however. You weren't even asked what you wanted the `root` password to be. The reason for this is because Ubuntu defaults to locking out the `root` account altogether. This is very much unlike many other distributions. In fact, Debian (on which Ubuntu is based), even has

you set a `root` password during installation. Ubuntu just decides to do things a little bit differently. 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, Ubuntu (as well as its server version), recommends the use of `sudo` rather than using `root` outright. 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've been able to do. For example, you cannot issue a command such as the following to install a software package as a normal user:

```
| apt install tmux
```

Instead, you'll receive an error:

```
| E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)
| E: Unable to lock the administration directory (/var/lib/dpkg/), 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. 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 for administering 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 either 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.

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

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

In the previous example, 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 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`).

If you list the storage of `/home`, you should see a folder listed there for our new user:



```
| ls -l /home
```

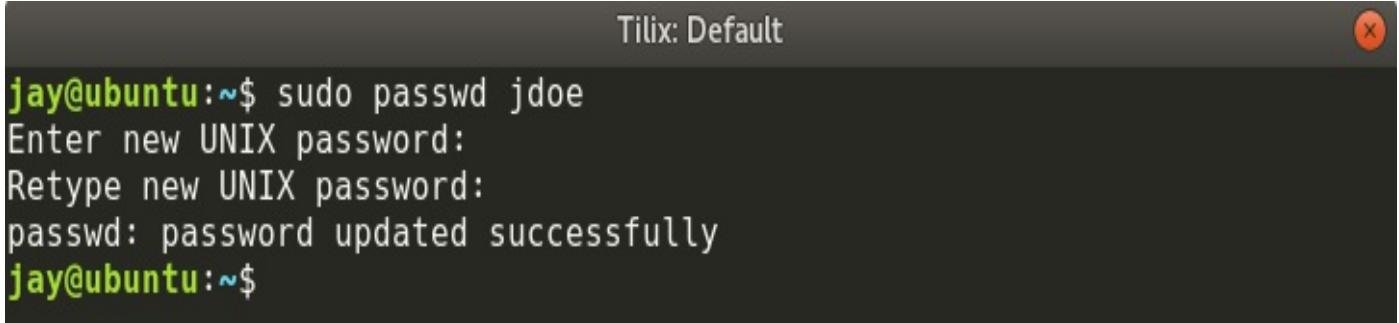
Tilix: Default

```
jay@ubuntu:~$ ls -l /home
total 8
drwxr-xr-x 3 jay jay 4096 Jan 14 09:34 jay
drwxr-xr-x 2 jdoe jdoe 4096 Jan 14 10:59 jdoe
jay@ubuntu:~$
```

Listing the contents of `/home` after our first user was created

What about creating our user's password? We created a new user on our system, but we did not set a password. We weren't even asked for one, what gives? 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:



```
Tilix: Default
jay@ubuntu:~$ sudo passwd jdoe
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
jay@ubuntu:~$
```

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 as you type a password while 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.

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 of a run of this process is as follows:

```
Tilix: Default
jay@ubuntu:~$ sudo adduser dscully
[sudo] password for jay:
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' ...
Enter new UNIX password:
Retype new UNIX 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:~$
```

Creating a user with the adduser command

In the previous 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 a 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 you to 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, we cover scripting in [Chapter 11](#), Learning Advanced Shell Techniques. In the case of `adduser`, it's a script written in Perl. Since it's not binary, you can even open it in a text editor in order to view all the magic code that it executes behind the scenes. However, make sure you don't open the file in a text editor with `root` privileges, so you don't accidentally save changes to the file and break it. 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 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 using `useradd` either directly or indirectly.

Now that we know how to create users, it will be useful to understand how to remove them as well. After all, 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 new hire will need access to the former user's files 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:

```
| ls -l /home
```

Tilix: Default

```
jay@ubuntu:~$ sudo userdel dscully
jay@ubuntu:~$ ls -l /home
total 12
drwxr-xr-x 2 1002 1002 4096 Jan 14 11:36 dscully
drwxr-xr-x 3 jay jay 4096 Jan 14 09:34 jay
drwxr-xr-x 2 jdoe jdoe 4096 Jan 14 10:59 jdoe
jay@ubuntu:~$
```

The home directory for 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 can 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 reside, 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 your logged-in user has access to. The following command will create the `file_archive` directory I mentioned in the following 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 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. For 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!

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



```
Tilix: Default  
lxr:x:106:65534::/var/lib/lxd:/bin/false  
uuidd:x:107:111::/run/uuid:/usr/sbin/nologin  
dnsmasq:x:108:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin  
sshd:x:109:65534::/run/sshd:/usr/sbin/nologin  
pollinate:x:110:1::/var/cache/pollinate:/bin/false  
jay:x:1000:1000:Jay Sherman,,,:/home/jay:/bin/bash  
jdoe:x:1001:1001::/home/jdoe:  
jay@ubuntu:~$
```

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`, `jdoe`, and so on. 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 with 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 us 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.

In my case, the UID of each user is the same as their GID. This is just a coincidence on my system and 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 things 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'd 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 first and last names. In my example, the fifth field is blank for `jdoe`. 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 while you read 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 wanted 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, or if we're in the mood to pull a prank on someone.

With that out of the way, let's take a look at the `/etc/shadow` file. We can use `cat` to display the contents as 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
Tilix: Default
jay:$6$G18LsnI0$RA6lB5kFWMGNnuks771Xq4f5U01vyEQFEMRdjjqWhA1NyRd3LFmj9Wtp6pMaZ0Cb8KN0Sa6d1JIUXWkHsaqc5L:/:17545:0:99999:7:::
jdoe:$6$bCQ508s9$7nP2A0pw02Qvn0Nee0LnK5ScEv0WoBg7IRg9Cn1x5e8wKkLM8vri1e8lADYZt990Ka1w5eknpy7fx7pj3Yo.z1:17545:0:99999:7:::
jay@ubuntu:~$
```

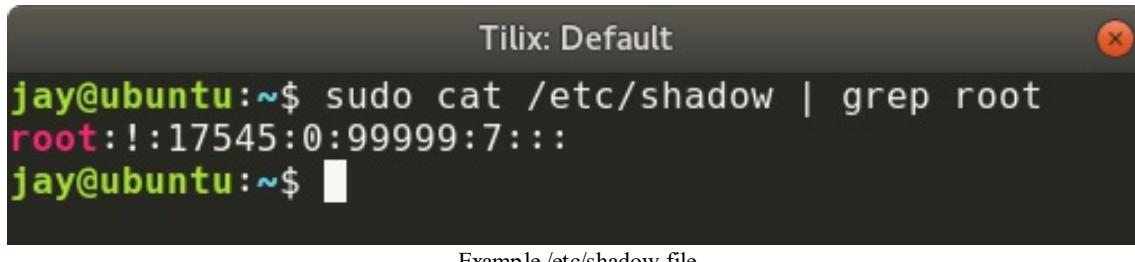
Example `/etc/shadow` file

The previous screenshot shows the last two 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 gobbledegook. Actually, that's the most important part of this entire file. That's the actual hash for the user's password. If you recall, in the `/etc/passwd` file, each user listing had an `x` for the second column, and I mentioned I would explain it 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 are. In the days of old, you could actually store a user's password in `/etc/passwd`, but it's literally 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:![:17545:0:99999:7:::
jay@ubuntu:~$
```

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 exclamation point where the password hash would've been. In practice, placing an exclamation point here is one way to lock out an account, though the standard practice is to use an `x` in this field instead. But either way, we can still switch to the `root` account (which I'll show you how to do later on in this chapter). The restriction is that we can't directly log in as `root` from the shell or over the network. We have to log in to the system as a normal user account first.

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 to save you the trouble of flipping back:

```
|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 since 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>
```



By executing this command, you can view information about any account on your system. 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 the case of my previous sample, `mulder` can change the password anytime 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 his or her password. However, never

underestimate the laziness of your users. It's quite common for a user, when required to change his or her password, to change their password several times to satisfy history requirements, 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.

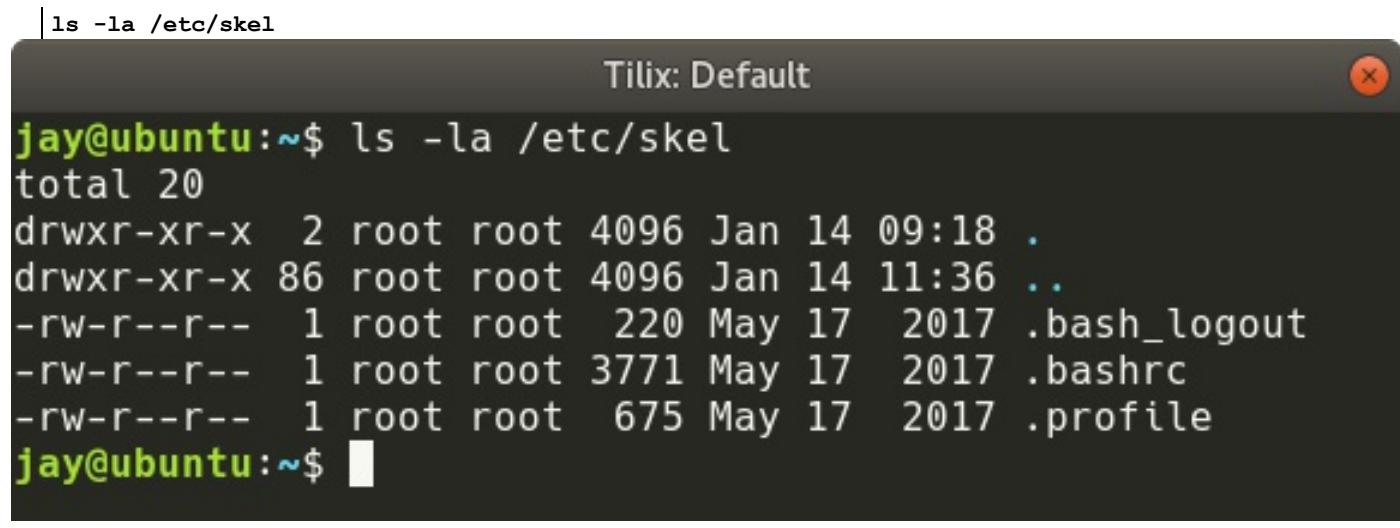
The fifth column, as you can probably guess, is the maximum number of days that can pass between password changes. If you require your users to change their passwords every certain number of days, you'll see that in this column. By default, this is set to 99,999 days. It probably stands to reason that you'll move on to bigger and better things by the time that elapses, 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 in 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. In our example, this is not set. Finally, with the eighth column, 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.

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. If you execute the following command, you can view the contents of the `/etc/skel` directory:



```
| ls -la /etc/skel
Tilix: Default
jay@ubuntu:~$ ls -la /etc/skel
total 20
drwxr-xr-x  2 root root 4096 Jan 14 09:18 .
drwxr-xr-x 86 root root 4096 Jan 14 11:36 ..
-rw-r--r--  1 root root  220 May 17 2017 .bash_logout
-rw-r--r--  1 root root 3771 May 17 2017 .bashrc
-rw-r--r--  1 root root  675 May 17 2017 .profile
jay@ubuntu:~$
```

Default /etc/skel files

You probably already know how to list files within a directory, but I specifically called out the `-a` parameter because the files included in `/etc/skel` by default are hidden (their filenames begin with a period). I threw in the `-l` parameter solely because I like it better (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 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 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 you favor 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 afterwards, and you'll see that these files will propagate into the home directories of new user accounts you add to your system.

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 providing 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 out the `root` account by default, so you really don't have a `root` password. 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 login 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 are now 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.

But 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. 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.

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, 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 a one-to-one ownership: just one user and just one group assigned to each file or directory. If you list contents of a directory on a Linux system, you can see this for yourself:

```
| ls -l
```

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

```
| -rw-r--r-- 1 root bind 490 2013-04-15 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. The `groups` command will tell you what groups your currently 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.

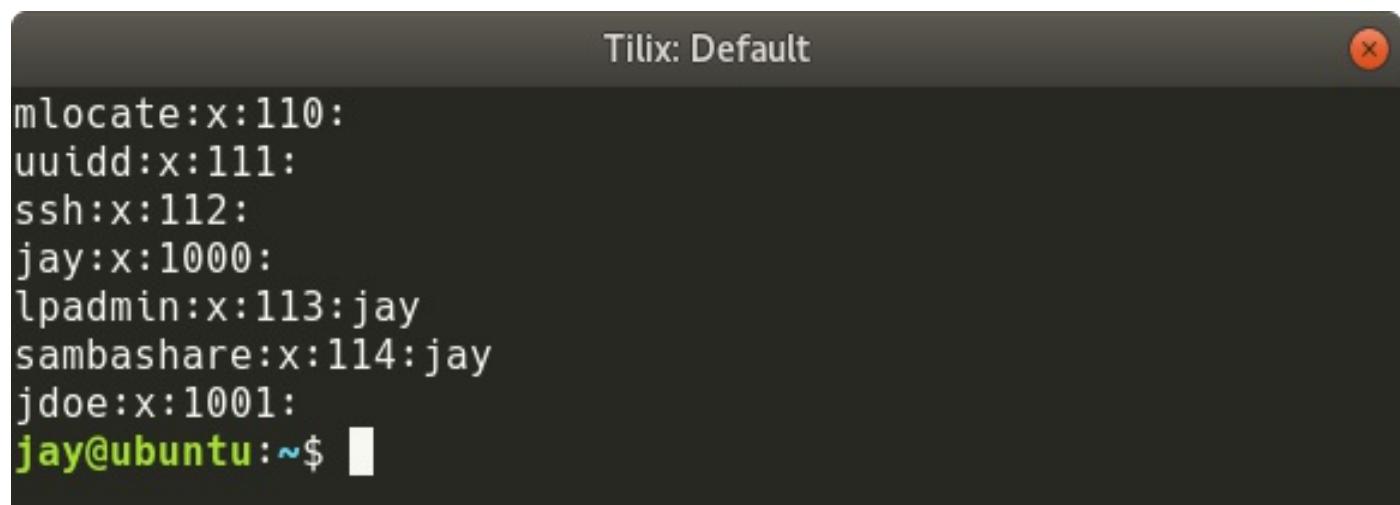
On the Ubuntu Server platform, you'll likely see that each of your user accounts is a member of just one group, a group that's named the same as your username. As I mentioned earlier, when you create a user account, you're also creating a group with the same name as the user. On some Linux distributions, 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.

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:



```
mlocate:x:110:  
uidd:x:111:  
ssh:x:112:  
jay:x:1000:  
lpadmin:x:113:jay  
sambashare:x:114:jay  
jdoe:x:1001:  
jay@ubuntu:~$
```

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. In the third column, we have the GID, which is similar in concept to the UID when we were discussing users. Finally, in the last column, we see a comma-separated list of each user that is a member of each of the groups. In this case, we're seeing that user `jay` is a member of the `lpadmin` and `sambashare` groups.

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 `/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 sales-people. 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 (though it's considered bad practice to do so), 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 thing it can do). 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 manual 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 move 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 efficient management of groups, you'll be able to best manage the resources on your server. 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 you'll 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 user name) allows us to change the password for any user account on our system. But that's not all this command can do.

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 lower your attack surface. To lock an account, use the `-l` option:

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

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

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

However, locking an account will not prevent the user from logging in if he or she has access to the server via SSH with 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 15](#), Securing Your Server. For now, just keep in mind that you can use the `passwd` to lock or unlock accounts, and if you utilize SSH, you'll want to lock your users out of that if you don't want them logging in.

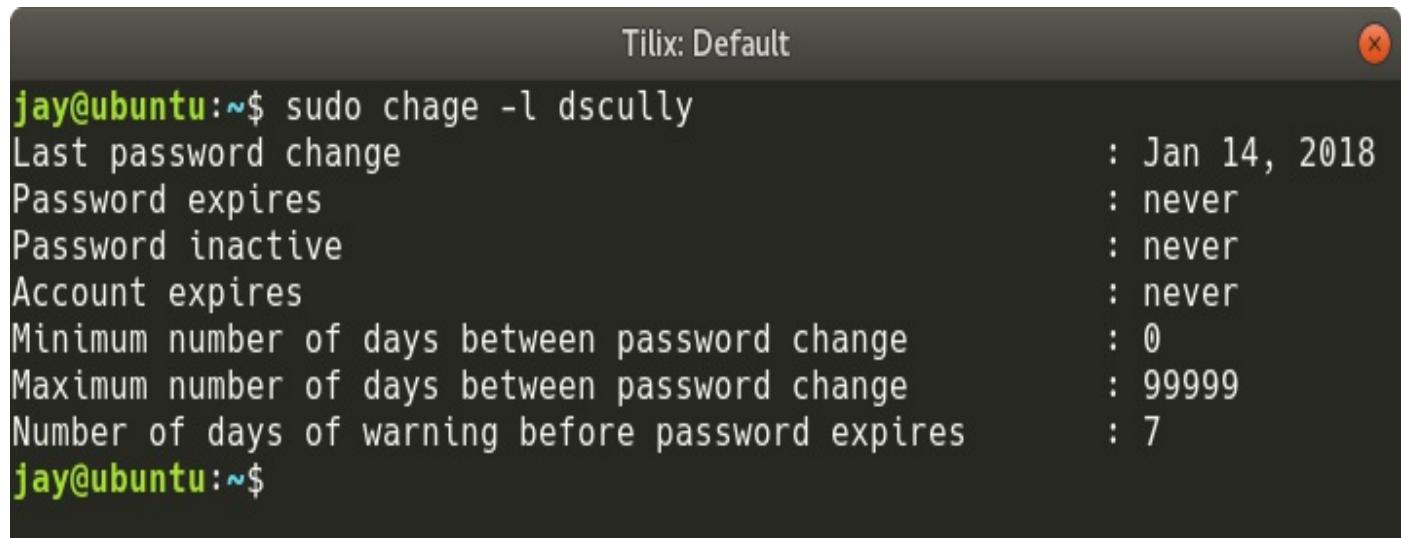
However, there's more to password management than the `passwd` command, as we can also implement our own policies as well. Earlier, I mentioned that you can set an expiration date on a user's password (during our discussion on the `/etc/shadow` file). In this section, we'll go through how to actually do that. Specifically, the `chage` command gives us this ability. We can use `chage` to not only 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:



```
Tilix: Default
jay@ubuntu:~$ sudo chage -l dscully
Last password change : Jan 14, 2018
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:~$
```

Output from the `chage` command

In the output, we can see values for the date of expiration, 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 to do 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` with the following example:

| `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>`

```
Tilix: Default
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:~$
```

The chage command listing a user that has a required password change set

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

```
| sudo chage -m 5 dscurly
```

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.

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 configure options for password requirements, let's first install the required **Pluggable Authentication Module (PAM)**:

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

Next, let's take a look at the following file. 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 your users can not only log in, but also your admins as well. 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 line 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 his or her 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 all comes down to it, use your best judgment.

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 `root`. 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 into the system. Using `sudo` is an alternative to using `root`, so you can give your administrators access to perform `root` tasks 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` 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 with `sudo` 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>`

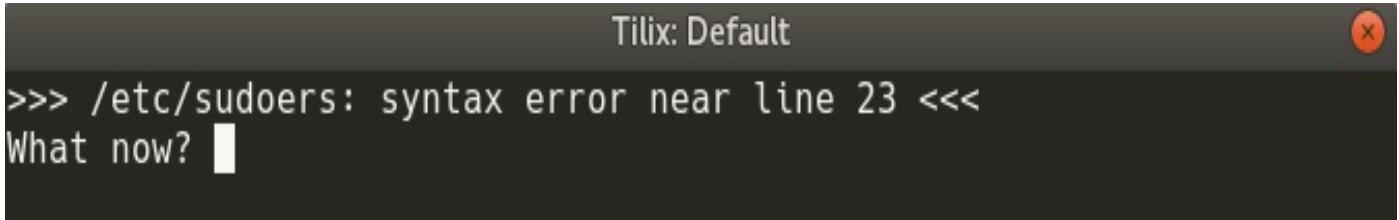


Not all distributions utilize the `sudo` group by default, or even automatically install `sudo`. Other distributions require you to install `sudo` manually and may use another group (such as `wheel`) to govern access to `sudo`.

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 thing, because if you did make some 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 23 <<<
What now? █
```

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`. However, you don't have to use `nano` if you prefer something else. If you fancy the `vim` text editor, you can use the following command to use `visudo` with `vim`:

```
| # sudo EDITOR=vim visudo
```

Anyway, 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 log off; otherwise, 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. 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), rather than 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)  /usr/sbin/reboot,/usr/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), he will receive an error message:

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

However, if `charlie` wants to use the `reboot` or `shutdown` commands on the server, he 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-server` 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-server=(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 version. 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`. He can use `apt update`, `apt dist-upgrade`, and any other sub-command of `apt`. But if he tries to reboot the server, remove protected files, add users, or anything else we haven't explicitly set, he 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-server= /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-server=(dscully:admins) ALL
```

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

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.

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 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--`). We also see the link count for the object (second column), the user that owns the file (third), the group that owns the file (fourth), the size in bytes (fifth), the last date the file was modified (sixth), and finally 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
-	r w-	r w-	r w-
-	r w-	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 the 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 are files, because the first position 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 group, we have three characters, `r w-`, `r w-`, and `rwx` respectively, in the second column of each object. 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 field shows us that `doctor` is the user that owns the file. Therefore, the second column of the permission string (`r w-`) applies specifically to the user `doctor`. Moving on, the third column of permissions is also three characters. In this case, `r w-` 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 string. Finally, the last portion of the permission string (`r w-` 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 `r w-` permissions on the object.

Individually, `r` stands for read and `w` stands for write. Therefore, we can read the second

column (`rw-`), indicating 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 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 other). 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 have 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 <code>cd</code> 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 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). In regard to 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.

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`) have their 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. 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 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 it. The second example

is locating all directories in this same path and changing them all to permissions 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.

Finally, we'll need to know how to change 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 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 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 chown 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 the field, 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 storage. It's going to be awesome.

Questions

1. Which command should you place in front of commands that require `root` privileges?
2. Name at least one of the two commands you can use to create a new user on Ubuntu Server.
3. Which command can you use to remove a user?
4. Which two files on the Linux filesystem store information regarding user accounts?
5. What is the name of the directory that stores default configuration files for users?
6. Assuming you have a user named `jdoe`, what command would you type to switch to that user?
7. Assuming you want to create a group named accounting what command would you use to accomplish that?
8. To add an expiration date to a password, you would use the _____ command.
9. You should use the _____ command to give a user access to `sudo`.
10. The _____ command allows you to change permissions of a file or directory, and the _____ command allows you to change its ownership.

Further reading

- User management (Ubuntu documentation): <https://help.ubuntu.com/lts/serverguide/user-management.html>
- File permissions (Ubuntu community wiki): <https://help.ubuntu.com/community/FilePermissions>

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:

- Understanding the Linux filesystem
- Using symbolic and hard links
- Viewing disk usage
- Adding additional storage volumes
- Mounting and unmounting volumes
- Partitioning and formatting volumes
- Understanding the `/etc/fstab` file
- Managing swap
- Utilizing LVM volumes
- Understanding RAID

Understanding the Linux filesystem

Before we get into the subject of managing storage volumes, we'll first need to have a better understanding of how the filesystem is laid out. 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 (ext4, XFS, and so on). In this section, we're going to take a quick look at the default directory structure.

In Linux (Ubuntu uses the Linux kernel and related utilities) 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 directory `jay`, which is inside the directory `home`, 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. 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 directory `home` among the results, among many others.

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 relative to managing storage volumes? When you add a volume to your server (which can be a physical disk in the case of a physical server, or a virtual disk when it comes to virtual machines) you format that volume, and then mount (attach) the volume to a directory on the filesystem. The directory you attach the volume to can have any name and be located anywhere you wish. The FHS defines specific directories for this purpose, but whether or not you follow that is up to you. You could certainly mount an additional drive to a directory called `kittens` located at the root of the filesystem, there's nothing stopping you. The question is, should you? That's what the FHS was defined to address.

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
/home	User home directories
/root	The home directory for <code>root</code> (<code>root</code> doesn't have a directory underneath <code>/home</code>)
/media	For removable media, such as flash drives
/mnt	For volumes that are intended to stay mounted for a while
/opt	Additional software packages (some programs are installed here, not as common)
/bin	Essential user binaries (<code>ls</code> , <code>cp</code> , and so on)
/proc	Virtual filesystem for OS-level components
/usr/bin	A majority of user commands

/usr/lib	Libraries
/usr/lib	Libraries
/var/log	Log files

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. When we get to the section of mounting volumes later on in this chapter, we'll use the filesystem directly.

Using symbolic and hard links

With Linux, we can link files to other files, which gives us quite a bit of flexibility with how we can manage our data. 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 are 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 are details such as the owner of the file, permissions, last modified date, and type (whether it is a directory or a file). But as a refresher, an inode is a data object that contains metadata regarding files within your filesystem. 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 `4456458` and `4456459` 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:~$ ls -i
4456458 file1 4456459 file2
jay@ubuntu:~$
```

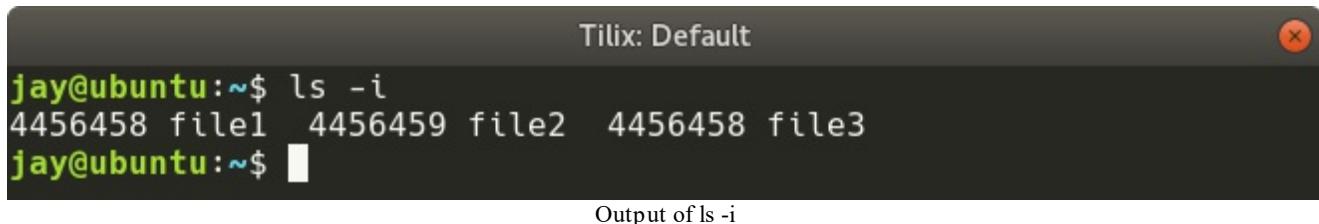
Output of `ls -i`

There are two types of links in Linux: symbolic links and hard links. This concept is similar in purpose to shortcuts created in graphical user interfaces. Almost all graphical operating systems have a means of creating a shortcut icon that points to another file or application. I'm sure you've seen shortcut icons to applications on Windows or macOS systems. Even Linux has this same functionality in most desktop environments that are available. On a Linux server, you typically don't have a graphical environment, but you can still link files to one another using symbolic or hard links. And while links approach the concept of shortcuts 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:



```
Tilix: Default
jay@ubuntu:~$ ls -i
4456458 file1 4456459 file2 4456458 file3
jay@ubuntu:~$
```

Output of `ls -i`

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 `4456458` 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 symbolic link instead. A symbolic link (also known as **soft links** or **symlinks**) 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 (as these do not share the same inode number as the original file), 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) 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:~$ ls -i  
4456458 file1 4456459 file2 4456460 file3  
jay@ubuntu:~$
```

Output of `ls -i` after creating a symbolic link

That's not the only thing that's different in the output, though. Notice that the inode numbers of each file are all different. At this point, the main difference from 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.

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 merely a pointer-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 are 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.

Viewing disk usage

Keeping tabs on your storage is always important, as no one enjoys getting a call in the middle of the night that a server is having an issue, much less something that could've been avoided, such as a filesystem growing too close to being full. Managing storage on Linux systems is easy once you master the related tools, the most useful of which I'll go over in this section. In particular, we'll answer the question ""what's eating all my free space?"" and I'll provide you with some examples of how to find out.

First, the `df` command. This command is likely always going to be your starting point in situations where you don't already know which volume 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 with `df` so that you'll see information that's a bit easier to read. Go ahead and give it a try:

```
| df -h
Tilix: Default
10:14:12 [0] [unicorn:~] % df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            3.9G   0    3.9G  0% /dev
tmpfs           785M  66M  720M  9% /run
/dev/mapper/vg_unicorn-lv_root  28G  6.5G  20G  25% /
tmpfs           3.9G  8.0K  3.9G  1% /dev/shm
tmpfs           5.0M   0    5.0M  0% /run/lock
tmpfs           3.9G   0    3.9G  0% /sys/fs/cgroup
/dev/mapper/vg_unicorn-lv_home  559G 376G 183G 68% /home
/dev/mapper/vg_unicorn-lv_store  1.9T 1.1T 750G 60% /store
/dev/mapper/backup_drive       3.6T 1.5T 2.0T 43% /media/backup_drive
tmpfs           785M   0    785M  0% /run/user/1000
10:14:51 [0] [unicorn:~] %
```

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 example I showed earlier, I took the information from one of my personal servers. In my case, you'll see that the root filesystem is located on `/dev/mapper/vg_unicorn-lv_root`. We know this because Mounted on is set to `/` (more on that later). In my case, this is an LVM volume, and we will talk about such volumes later in this chapter. The name of the volume group, `vg_unicorn-lv_root` is arbitrary, names of LVM volumes are provided by you and I just named mine after the hostname of the server (`unicorn`). On your side, you'll probably just see this as `/dev/sda1`, or similar. On non-LVM systems, this name differs quite a bit depending on what type of disk you have. Furthermore, we can also glean from this output that only 1% of this volume is used up, and the volumes mounted at `/home` and `/store` are 68% and 60% utilized respectively. In general, you can see from the example, the `df` command gives you some very useful information for evaluating your current storage usage. It shows you the filesystem, its size, how much space is used, how much is available, how much percentage is being used (used percentage), and the filesystem the device is mounted on.

While investigating disk utilization, it's also important to check inode utilization. Checking inode utilization will be especially helpful in situations where an application is reporting that your disk is full but the `df -h` command shows plenty of free space is available. 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).

The problem with inodes is that you can only have a limited number of them on any storage media. This number is usually extremely high and hard to reach. In the case of servers, though, where you're possibly dealing with 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
```

Tilix: Default

```
10:35:34 [0] [unicorn:~] % df -i
Filesystem           Inodes   IUsed   IFree  IUse% Mounted on
udev                 999996    501    999495    1% /dev
tmpfs                1004799   724    1004075    1% /run
/dev/mapper/vg_unicorn-lv_root  1831424 162571   1668853    9% /
tmpfs                1004799     3    1004796    1% /dev/shm
tmpfs                1004799     4    1004795    1% /run/lock
tmpfs                1004799    16    1004783    1% /sys/fs/cgroup
/dev/mapper/vg_unicorn-lv_home 292968448 50075  292918373    1% /home
/dev/mapper/vg_unicorn-lv_store 195312000  3289  195308711    1% /store
/dev/mapper/backup_drive      244195328 41802  244153526    1% /media/backup_drive
tmpfs                1004799     4    1004795    1% /run/user/1000
10:35:53 [0] [unicorn:~] %
```

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, my root filesystem has a total of `1668853` inodes available, of which `162571` are used and `1831424` 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 limit 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.

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 are 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 a total amount of space used within our current working directory. The following screenshot shows this output from my laptop:

```
10:51:11 [0] [seraph:~] % du -hsc *
0          desktop
1.4G      downloads
816K      git
4.1G      nextcloud
13M       snap
5.5G      total
10:51:21 [0] [seraph:~] %
```

Example output from `du -hsc *`

As you can see, the information provided by `du -hsc *` is a nice, concise summary. From the output, we know which directories within our working directory are the largest. To further narrow down our storage woes, we could `cd` into any of those large directories and run the 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` command). 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:

Tilix: Default

```
ncdu 1.12 ~ Use the arrow keys to navigate, press ? for help
-- / --
4.7 GiB [#####] /usr
1.4 GiB [##] swapfile
1.0 GiB [##] /var
710.5 MiB [#] /lib
132.5 MiB [ ] /boot
19.9 MiB [ ] /etc
15.4 MiB [ ] /sbin
11.9 MiB [ ] /bin
1.3 MiB [ ] /tmp
48.0 KiB [ ] /root
20.0 KiB [ ] /snap
e 16.0 KiB [ ] /lost+found
8.0 KiB [ ] /media
4.0 KiB [ ] /lib64
Total disk usage: 8.0 GiB Apparent size: 8.0 GiB Items: 255572
```

ncdu in action

To operate `ncdu`, you move 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!

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.

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. The `fdisk` command is normally used for creating and deleting partitions, but it will allow us to determine which device name our new disk received. Basically, the `fdisk -l` command will give you the info, but you'll need to run it as `root` or with `sudo`:

```
| sudo fdisk -l
Tilix: Default
Disk /dev/vda: 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
Disklabel type: dos
Disk identifier: 0x1a7d4c6a

Device      Boot Start      End  Sectors Size Id Type
/dev/vdal   *     2048 41943006 41940959  20G 83 Linux
15:37:10 [0] [web:~] %
```

Output of the `fdisk -l` command, showing a volume of `/dev/vdal`

I always recommend running this command before and after attaching a new device. That way, it will be obvious which device name is the one that's new. Once you have the name of the device, we will be able to interact with it and set it up. 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've attached it, what device name did it receive? As I mentioned, run the `fdisk -l` command as `root` to find out. Another trick is to use the following command, with which the output will update automatically as you add the disk. Just start the command, attach the disk, and watch the output. When done, press `Ctrl + C` on your keyboard:

| `dmesg --follow`

- How do you want it 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, which you may also consider for additional volumes. We'll discuss formatting later in this chapter.



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 later on in this chapter, I'll discuss a few common locations.

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 LVM, 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 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.

The device name, as we discussed, is found with the `fdisk -l` command. 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. Either works fine:

```
| lsblk
Tilix: Default
15:44:48 [0] [git:~] % lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda    253:0    0  20G  0 disk
└─vda1  253:1    0 19.9G  0 part /
  └─vda14 253:14   0    4M  0 part
  └─vda15 253:15   0 106M  0 part /boot/efi
loop0   7:0      0 83.8M  1 loop /snap/core/3748
loop1   7:1      0 83.7M  1 loop /snap/core/3440
loop2   7:2      0 83.8M  1 loop /snap/core/3604
15:45:04 [0] [git:~] %
```

Output of the `lsblk` command, showing a virtual disk on a DigitalOcean droplet

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). You'll also need to know the partition number as well. 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 RAID controller or 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.

Next, you need to consider which filesystem to use. `ext4` is the most common filesystem type but many others exist, and new ones are constantly being created. At the time of writing, there are up and coming filesystems such as **B-tree file system (Btrfs)** being developed, while `ZFS` is not actually new but is new to Ubuntu (it's very common in the BSD community and has been around for a long time). Neither `Btrfs` nor `ZFS` are

considered ready for stable use in Linux due to the fact that Btrfs is relatively new and ZFS is newly implemented in Ubuntu. At this point, it's believed that ext4 won't be the default filesystem forever, as several are vying for the crown of becoming its successor. As I write this, though, I don't believe that ext4 will be going away anytime soon and it's a great default if you have no other preference.

An in-depth look at all the filesystem types is beyond the scope of this book due to the sheer number of them. One of the most common alternatives to ext4 is XFS, which is a great filesystem if you plan on dealing with very large individual files or massive multi-terabyte volumes. XFS volumes are also really good for database servers due to their additional performance. In general, stick with ext4 unless you have a very specific use case that gives you a reason to explore alternatives.

Finally, the process of adding a new volume entails determining where you want to mount it and adding the device to the `/etc/fstab` file, which will help with automatically mounting it each time the server is booted (which is convenient but optional). We'll discuss the `/etc/fstab` file and mounting volumes in a later section. Basically, Linux volumes are typically mounted inside an existing directory on the filesystem, and from that point on, you'll see free space for that volume listed when you execute the `df -h` command we worked through earlier.

It's very typical to use the `/mnt` directory as a top-level place to mount additional drives. If you don't have an application that requires a disk to be mounted in a specific place, a sub-directory of `/mnt` is a reasonable choice. I've also seen administrators make their own top-level directory, such as `/store`, to house their drives. When a new volume is added, a new directory would be created underneath the top-level directory. For example, you could have a backup disk attached to `/mnt/backup` or a file archive mounted at `/store/archive`. The directory scheme you choose to use is entirely up to you.

I just mentioned mounting a backup disk at `/mnt/backup`, but I want to be clear that doesn't necessarily imply an internal disk. In Linux, you can mount storage volumes and disks, but you can also mount external resources as well, such as external hard disks or network shares. Therefore, `/mnt/backup` may be a network location on a remote server, or an internal or external disk. When it comes to different storage types, the only real difference is how they're mounted and formatted. But once the volume is set up, it's treated the same as any other from that point forward. Mounting volumes will be discussed later on in this chapter.

Partitioning and formatting volumes

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 be partitioned and formatted. We used the `fdisk` command earlier to see a list of current partitions on our disk, but this command does much more than just show us what partitions are available, it allows us to manage them as well. In this section, I'll walk you through partitioning as well as formatting new volumes.

In order to begin the process of partitioning a disk, we would first determine the naming designation the disk received using either the `lsblk` or `sudo fdisk -l` commands as we've done earlier. In my case, I added a new disk to my sample server, and using `fdisk -l`, it's pretty easy to see which disk is the new one:

```
| sudo fdisk -l
Tilix: Default
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: dos
Disk identifier: 0x50af5128

Device      Boot   Start     End   Sectors   Size Id Type
/dev/sdal    *       2048  58593279  58591232   28G 83 Linux
/dev/sda2        58593280 335542271 276948992 132.1G 83 Linux

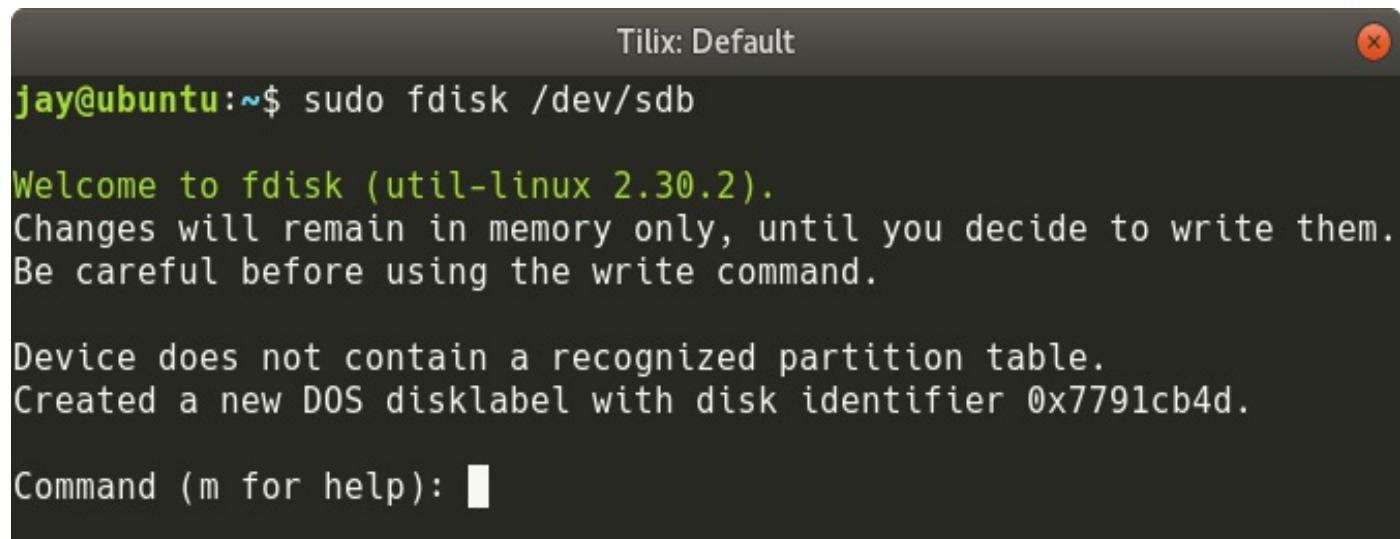
Disk /dev/sdb: 30 GiB, 32212254720 bytes, 62914560 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
jay@ubuntu:~$
```

Beginning the process of adding a new storage volume to a virtual server

In this case, we can see the `/dev/sdb` has no partitions (and this volume wasn't present before I added it), so we know what disk we need to work on. Next, 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:



```
Tilix: Default
jay@ubuntu:~$ sudo fdisk /dev/sdb

Welcome to fdisk (util-linux 2.30.2).
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 0x7791cb4d.

Command (m for help):
```

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 I have to make sure you're 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 make sure 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 **MBR** and **GPT** partition tables. When creating partitions, you'll have the option to use an MBR partition table or a 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. By default, `fdisk` will create a partition table in the MBR format. But 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. 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. It's certainly no wonder why GPT is fast becoming the new standard! It's only a matter of time before GPT becomes the default, so unless you have good reason not to, I recommend using it if you have a choice.

When you first enter the `fdisk` prompt, you can press `m` to access the menu, where you'll see that you have a few options toward the bottom for the partition style you'd like to use. If you make no selection and continue with the process of creating a partition, it will default to MBR. Instead, you can press `g` at the prompt to specifically create a GPT partition table, or to switch back to MBR. Note that this will obviously wipe out the current partition table on the drive, so hopefully you weren't storing anything important on the disk.

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. Afterwards, you'll be asked for the first sector for the partition to use (accept the default of 2,048) and then 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. If you'd like to create multiple partitions, don't accept the default at this prompt. Instead, you can clarify the size of your new partition by giving it the number of megabytes or gigabytes to use. For example, you can enter `20G` here to create a partition of 20 GB.

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:

Tilix: Default

```
jay@ubuntu:~$ sudo fdisk /dev/sdb

Welcome to fdisk (util-linux 2.30.2).
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 0x7791cb4d.

Command (m for help): n
Partition type
  p  primary (0 primary, 0 extended, 4 free)
  e  extended (container for logical partitions)
Select (default p):

Using default response p.
Partition number (1-4, default 1):
First sector (2048-62914559, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-62914559, default 62914559): +20G

Created a new partition 1 of type 'Linux' 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:~$
```

Example run of the fdisk command

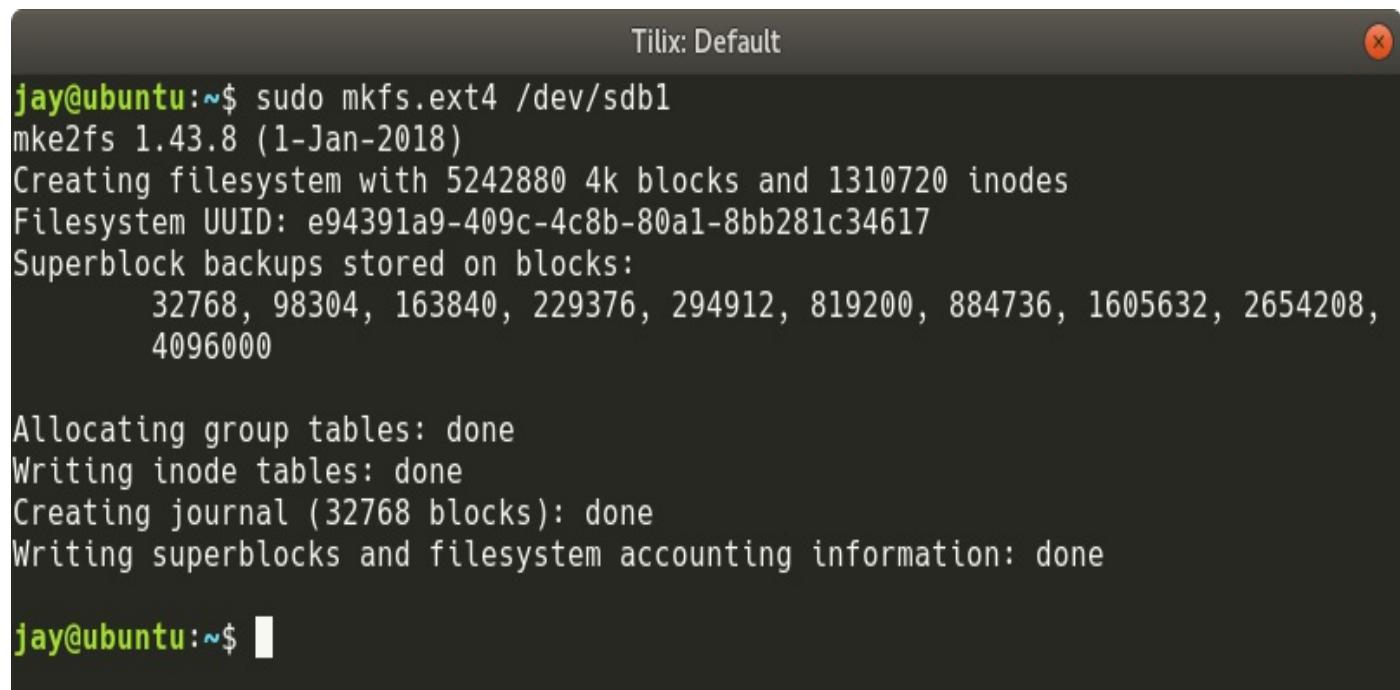
After you create your partition layout for your new disk and you're satisfied with it, you're ready to format it. If you've made a mistake or you want to redo your partition

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 is done with the `mkfs` command. To format a device, you execute `mkfs` 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:



```
Tilix: Default
jay@ubuntu:~$ sudo mkfs.ext4 /dev/sdb1
mke2fs 1.43.8 (1-Jan-2018)
Creating filesystem with 5242880 4k blocks and 1310720 inodes
Filesystem UUID: e94391a9-409c-4c8b-80a1-8bb281c34617
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:~$
```

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 mfs.xfs /dev/sdb1
```

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 it to be mounted. 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 FHS that was mentioned earlier. The purpose 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 that your users and applications will use are mounted. 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 here.

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 when you've come up with such a place, 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`. I needed to have created the `/mnt/vol1` directory before this command could work, so I would've already created that directory with the following command:

```
| sudo mkdir /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:

```
| sudo 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 following command before and after mounting:
`df -h`



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 because 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.

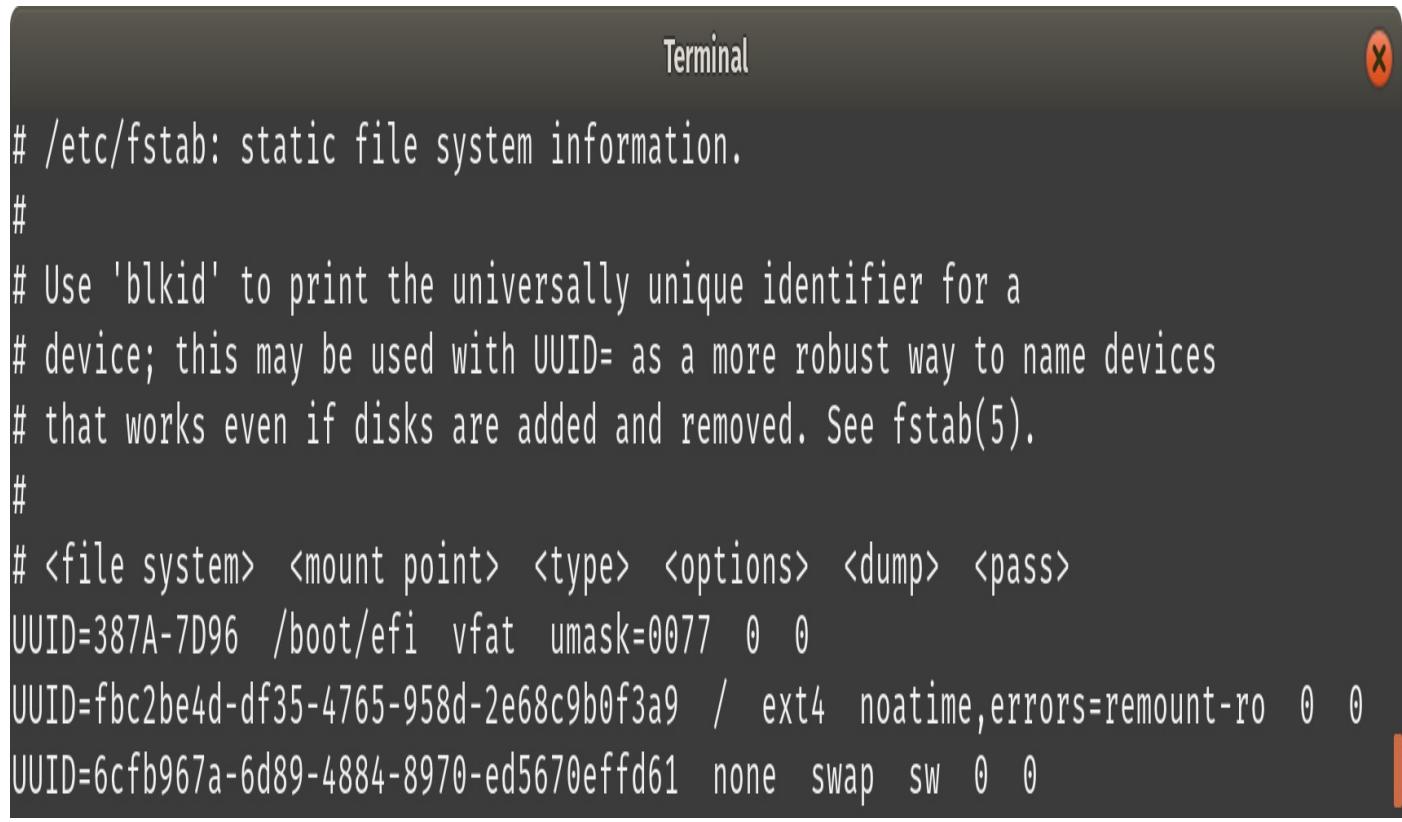
The downside to manually mounting devices is that they will not automatically remount themselves the next time your server boots. In order to make the mount automatically come up when the system is started, you'll need to edit the `/etc/fstab` file, which I'll walk you through in the next section.

Understanding the /etc/fstab file

The `/etc/fstab` file is a very critical file on your Linux system. As I mentioned in the last section, 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 as well, so if you make a mistake while editing it, your server will not boot. Definitely be careful.

When your system boots, it looks at this file to determine where the root filesystem is. In addition, the location of your `swap` area is also 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 mounts 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).

For an example, here is the content of `/etc/fstab` on one of my machines:



A screenshot of a terminal window titled "Terminal". The window shows the contents of the `/etc/fstab` file. The text is as follows:

```
# /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>
UUID=387A-7D96 /boot/efi vfat umask=0077 0 0
UUID=fbcb2be4d-df35-4765-958d-2e68c9b0f3a9 / ext4 noatime,errors=remount-ro 0 0
UUID=6cfb967a-6d89-4884-8970-ed5670effd61 none swap sw 0 0
```

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 you created during installation. On the server I used to grab the example `fstab` content, I have a single partition for the root filesystem, and I'm also mounting a `swap` file (we'll discuss `swap` later).

Each volume is designated with a **Universally Unique Identifier (UUID)** instead of the normal `/dev/sdax` naming convention you are more than likely used to. In my output, you can see that some comments (lines beginning with `#`) were created by the installer to let me know which volume refers to which device. For example, UUID `8d9756d9-c6f5-4efe-9d41-2d5e43c060c2` refers to my root filesystem, and you can also see that I have a swap file located at `/swapfile`.

The concept of a UUID has been around for a while, but there's nothing stopping you from replacing the UUID with the actual device names (`/dev/sda1` and `/dev/sda5` in my case). 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 (which SATA port, USB port, 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 receive. 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 where 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). 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. 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 second entry (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 being `ext4`, and the second being `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 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 are `0` and `0` in the first 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 second priority and checked last. As a general rule of thumb, use `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. First, we 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
```

The output of that command will give us the UUID. Copy that down in a text editor. Next, we need to know where we want to mount the volume. Go ahead and create the

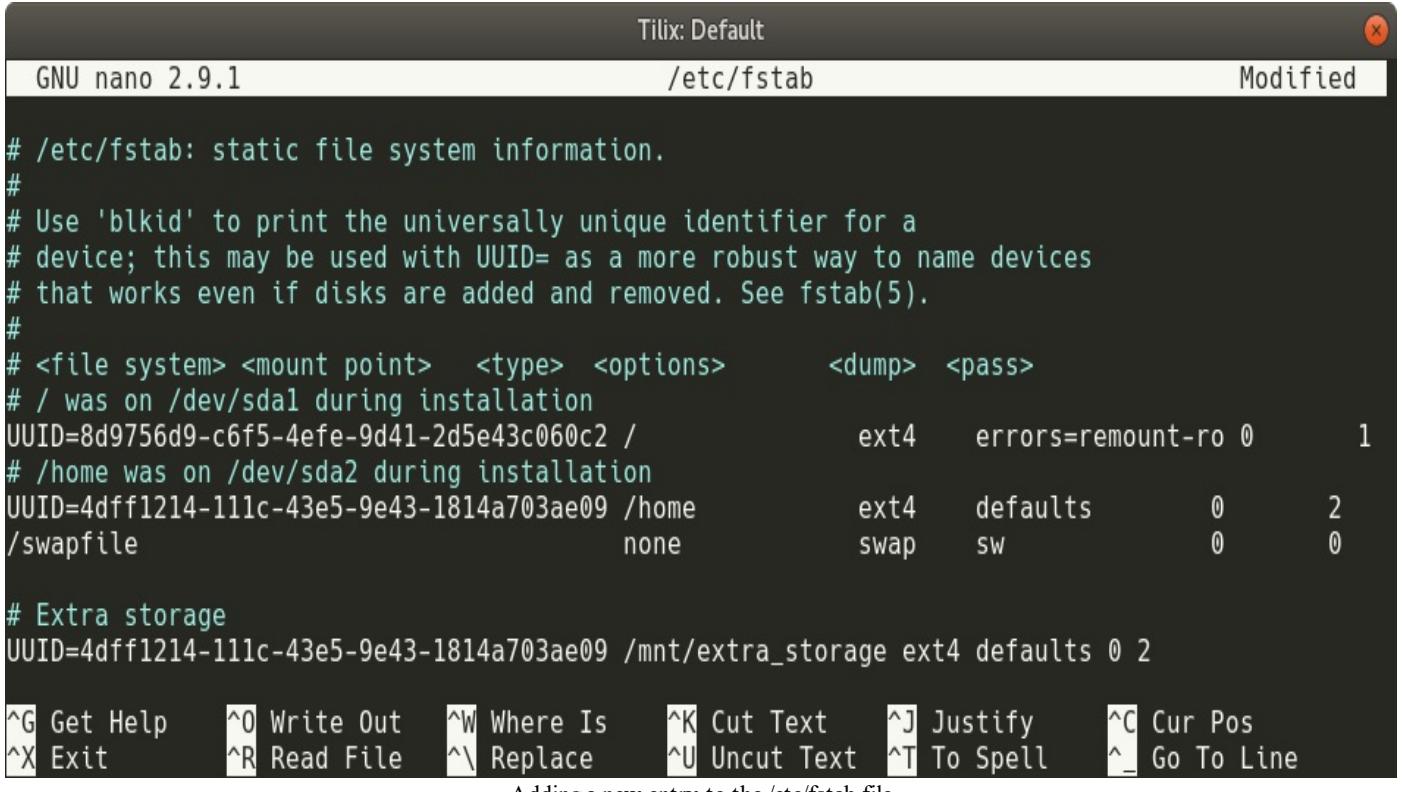
directory now, or use an existing directory if you wish. For example, you could create the directory `/mnt/extra_storage` for this purpose:

```
| 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, adding an entry for `/dev/sdb` would look similar to the following:



The screenshot shows a terminal window titled "Tilix: Default". The title bar includes "GNU nano 2.9.1", the file path "/etc/fstab", and the status "Modified". The main area of the window displays the contents of the /etc/fstab file. At the bottom, there is a menu bar with various keyboard shortcuts for nano, and a status message "Adding a new entry to the /etc/fstab file".

```
# /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/sda1 during installation
UUID=8d9756d9-c6f5-4efe-9d41-2d5e43c060c2 /          ext4    errors=remount-ro 0      1
# /home was on /dev/sda2 during installation
UUID=4dff1214-111c-43e5-9e43-1814a703ae09 /home        ext4    defaults        0      2
/swappfile           none            swap      sw        0      0

# Extra storage
UUID=4dff1214-111c-43e5-9e43-1814a703ae09 /mnt/extra_storage ext4 defaults 0 2
```

In my example, I created a comment line with a little note about what the extra volume will be used for. 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 `2`.

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 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 `fstab` just to use `noauto`. 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 with the following command:

```
| sudo mount /mnt/ext_disk
```

Notice that I didn't have to include the device name or options; only where I want the device mounted. 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 a list of everything that's mounted. Go ahead and try that, and I'll meet you in the next section.

Managing swap

Several times in this chapter, I mentioned the `swap` partition but have yet to give it a formal discussion. Swap is one of those things we never want to use, but always want to make sure is available. 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.

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` partition 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. It's a good idea to have it, and considering that hard drive space is cheaper nowadays, there's really no reason not to.

The way `swap` is implemented 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. If you need larger `swap`, you can delete the `swap` file and recreate it. That's definitely an easier thing to do than having to resize your partition tables 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.

Although I did say that you should definitely include a `swap` partition on your server, there is an exception to this rule, though. When it comes to the Raspberry Pi, you should never have `swap` space if you can help it. Traditionally, the Raspberry Pi images do not include a `swap` partition or file. You could create one, but I recommend that you don't. The reason for this is because the Pi uses SD cards for storage, which are orders of magnitude slower than traditional hard disks. While the Raspberry Pi might seem fast enough with regards to I/O when performing average tasks, `swap` would be so slow on these devices that it would just be more trouble than it's worth and end up working against you (your

Pi would start running so slow it would essentially be unresponsive). Since almost no one runs mission-critical workloads on the Raspberry Pi, the absence of a `swap` file shouldn't affect you on that platform.

As you probably noticed from our earlier discussion, the `swap` file is declared in our `/etc/fstab` file. 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 (which we will discuss later) and then use the `swapon` command to activate it:

```
| sudo swapon -a
```

Also, you'd create an entry in the `fstab` file for this `swap` file to ensure it gets activated every time the server is started.

When run, the `swapon -a` command will find your `swap` partition in `/etc/fstab`, mount it, and activate it for use. The inverse of this command is `swapoff -a`, which deactivates and unmounts your `swap` partition. 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.

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:

```
| 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`:

```
| 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, you have a handy-dandy `swap` file to use in a situation where your server runs out of memory. 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 as long as you created an adequate `swap` partition during installation. 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.

Utilizing LVM volumes

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 one of 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.

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.

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.

When setting up a new server via Ubuntu's alternative 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, but not

required. 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, each called a physical volume. Keep in mind that these disks are still referred to as physical volumes, 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 normal 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, you'll first need to 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:

```
| dpkg -s lvm2 | grep status
```

If it's not present (the output of the previous command doesn't come back as `install ok` `installed`, 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 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. The `pvcreate` command allows us to create physical volumes, so we'll need to run the `pvcreate` command against all of the drives we wish to use for this purpose. Since I have four, I'll use 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:

Tilix: Default

```
[sudo] password for jay:  
"/dev/sdb1" is a new physical volume of "20.00 GiB"  
--- NEW Physical volume ---  
PV Name          /dev/sdb1  
VG Name  
PV Size         20.00 GiB  
Allocatable      NO  
PE Size          0  
Total PE         0  
Free PE          0  
Allocated PE     0  
PV UUID          AG42hV-IsjR-DyeG-U2ll-rMxv-czBX-gtUHim  
  
jay@ubuntu:~$
```

Output of the `pvdisplay` command on a sample server

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/sdb1
```

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 to view details about it, including the number of assigned disks (which should now be 1):

Tilix: Default

```
jay@ubuntu:~$ sudo vgdisplay
--- Volume group ---
VG Name          vg-test
System ID
Format          lvm2
Metadata Areas   1
Metadata Sequence No  1
VG Access        read/write
VG Status        resizable
MAX LV           0
Cur LV           0
Open LV          0
Max PV           0
Cur PV           1
Act PV           1
VG Size          20.00 GiB
PE Size          4.00 MiB
Total PE         5119
Alloc PE / Size  0 / 0
Free  PE / Size  5119 / 20.00 GiB
VG UUID          6Ms5o6-aMJD-0cnV-oRPr-ymLh-g4kq-eRT89l
```

jay@ubuntu:~\$

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 create a logical volume and format it. Our volume group can contain all, or a portion, of the disk we've assigned to it. With the following command, I'll create a logical volume of 10 GB, out of the 20 GB disk I added to the volume group:

```
| sudo lvcreate -n myvol1 -L 10g vg-test
```

The command may look complicated, but it's not. In this example, I'm giving my logical volume a name of `myvol1` with the `-n` option. Since I only want to give it 10 GB of space, I use the `-L` option and then `10g` to represent 10 GB. Finally, I give the name of the volume group that this logical volume will be assigned to. You can run `lvdisplay` to see

information regarding this volume:

```
| sudo lvdisplay
jay@ubuntu:~$ sudo lvdisplay
--- Logical volume ---
LV Path          /dev/vg-test/myvol1
LV Name          myvol1
VG Name          vg-test
LV UUID          EsjutA-GG3r-gICp-rTTJ-0Dm0-8p1G-Y8yG5z
LV Write Access  read/write
LV Creation host, time  ubuntu, 2018-01-21 18:28:06 -0500
LV Status        available
# open           0
LV Size          10.00 GiB
Current LE       2560
Segments         1
Allocation       inherit
Read ahead sectors  auto
- currently set to 256
Block device     253:0
```

jay@ubuntu:~\$ █

Output of the `lvdisplay` command on a sample server

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 the previous screenshot under `LV Path`). Let's format it:

```
| 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 10 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 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
```

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. If you check your mounted disks with the `df -h` command, you should see this volume is mounted:

```
| 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 in 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.

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.

As you can see, LVM is very useful for managing storage on your server. It gives you the ability to scale your server's storage as the needs of your organizations and users evolve. However, what if I told you that being able to resize your storage on command isn't the only benefit of LVM? In fact, LVM also allows you to perform snapshots as well.

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. If you used Ubuntu's alternative installer, you were offered the option to create an LVM configuration during installation of Ubuntu Server, 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, 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.

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. If we would like to remove the 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 ummount and remount after merging. Afterwards, you'll see that the snapshot is removed the next time you run the `lvs` command.

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
```

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.

Understanding RAID

Now that we know the ins and outs of managing storage, we must face a simple and uncomfortable truth: disks fail. It's not a matter of if, but when—all disks will fail eventually. When they do, we rely on our backups and disaster recovery procedures to get up and running. One thing that can help us with this burden is **RAID**, which is an acronym for **Redundant Array of Inexpensive Disks**. The basic idea of RAID is that when a disk does fail, we don't lose any data (unless additional disks fail) and we'll continue on without any significant downtime assuming we replace the disk within a reasonable time frame. This isn't used as a backup, but it is a nice safety net in a situation where we lose a disk.

There are two types of RAID, **hardware RAID** and **software RAID**. With hardware RAID, the operating system is completely oblivious to the fact that RAID is even present. In this case, RAID is managed by a hardware controller card on the server, that completely abstracts the disk layout from the OS. Basically, the OS only sees one disk regardless of how many are on the controller card. With software RAID, the operating system manages the RAID configuration, and it does see all the disks. On a server with RAID already set up, it's very easy to see whether or not you're running with software or hardware RAID. Simply use the `sudo fdisk -l` command, which shows us the disks installed in the system. If it shows only one disk, it's hardware RAID. If it shows multiple, it's software.

Hardware RAID is beyond the scope of this book. Reason being, each controller card is different, and many servers have a different implementation so there is no way to cover them all. With hardware RAID, you'd want to consult the documentation that came with your server for information on how to configure it.

Software RAID is much more common with Linux, and is generally a better choice (though some administrators would argue the opposite, and they wouldn't necessarily be wrong depending on their environment). Most of the RAID controller cards out there are known as **fakeRAID**, which means that it attempts to operate as close to hardware RAID as possible, but at the end of the day it's easy to tell the difference (true hardware RAID means the server only sees one disk). What's worse is that some of these controller cards, even if they are true hardware RAID, do not ship drivers for Linux distributions, and may not function on anything other than Windows. The Linux implementation of software RAID is known as **MDRAID**, which we manage via the

`mdadm` command. Since this is a native Linux tool, it works very well for this purpose. (In case you're wondering, the `mdadm` command is an acronym for **Multiple Disk And Disk Administration**).

Another benefit of MDRAID is that we can easily use the `mdadm` command-line tool to query the status of the RAID array, to determine its health. We can also write scripts that will check the array periodically, and alert us if there is failure. Hardware RAID can certainly alert us as well, but native tools in a Linux installation are generally more reliable and this is yet another reason why I prefer this approach.

To implement MDRAID it's preferable to do so right at the beginning—and set it up while installing Ubuntu. For this, we need to use the alternative installer, because this feature is not available with the default Ubuntu installer. In the appendix at the end of the book, the alternative installer is covered. In addition, the setup process for RAID is covered there as well. If you feel you would benefit from RAID, consult the appendix to learn more.

Summary

Efficiently managing the storage of your servers will ensure that things continue to run smoothly, as a full filesystem is a sure-fire reason for everything 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, mount, and unmount volumes, as well as managing LVM, monitoring disk usage, and creating links. We also discussed `swap`, as well as managing our `/etc/fstab` file.

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.

Questions

1. Which command identifies the amount of space used in a directory?
2. The _____ is a specification that defines the default Linux filesystem directory structure.
3. Which command shows the total amount of disk space used for each mount point?
4. The _____ command allows you to attach an external storage resource (such as an external drive) to your local filesystem.
5. The _____ command allows you to partition your disk.
6. Which file allows you to automatically mount storage volumes at boot time?
7. A _____ file is utilized when system RAM is being heavily used.
8. The _____ command shows you a list of LVM logical volumes on your system.

Further reading

- **Filesystem Hierarchy Standard:** <http://www.pathname.com/fhs/>
- **Linux RAID reference:** https://raid.wiki.kernel.org/index.php/RAID_setup
- **Ubuntu LVM documentation:** <https://wiki.ubuntu.com/Lvm>

Connecting to Networks

Linux networks are taking the industry by storm. Many big-name companies use Linux in their data centers, to the point where most people use Linux nowadays, whether they realize it or not (either directly or indirectly). The scalability of Linux in the data center lends itself very well to networking. This flexibility allows Linux to not only be useful for large server deployments, but also allows it to power routers and network services.

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 connecting to other nodes and networks. We'll resume this exploration in [Chapter 7](#), Setting Up Network Services, where we'll work on some foundational concepts that power many of the things we'll need to set up our Linux network.

In this episode of our Ubuntu adventure, we will cover:

- Setting the hostname
- Managing network interfaces
- Assigning static IP addresses
- Understanding NetworkManager
- Understanding Linux name resolution
- Getting started with OpenSSH
- Getting started with SSH key management
- Simplifying SSH connections with a config file

Setting the hostname

During installation, you were asked to create a hostname for your server. The default during installation is `ubuntu`, but you can (and should) come up with your own name. If you left the default, or if you want to practice changing it, we'll work through that in this section.

In most organizations, there is a specific naming scheme in place for servers and networked devices. 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.

Your hostname identifies your server to the rest of the network. While the default `ubuntu` hostname 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 one another. But there's more to a server's name than its hostname, which we'll get into in [chapter 7, 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:

```
Tilix: Default
jay@ubuntu:~$ hostname
dev.mynetwork.org
jay@ubuntu:~$
```

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.



The `hostnamectl` command didn't exist before Ubuntu switched to `systemd` (versions 15.04 and earlier). If you're using a version earlier than that, you'll need to `edit /etc/hostname` manually.

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 a typical server:

```
Tilix: Default
jay@ubuntu:~$ cat /etc/hosts
127.0.0.1      localhost
127.0.1.1      dev.mynetwork.org

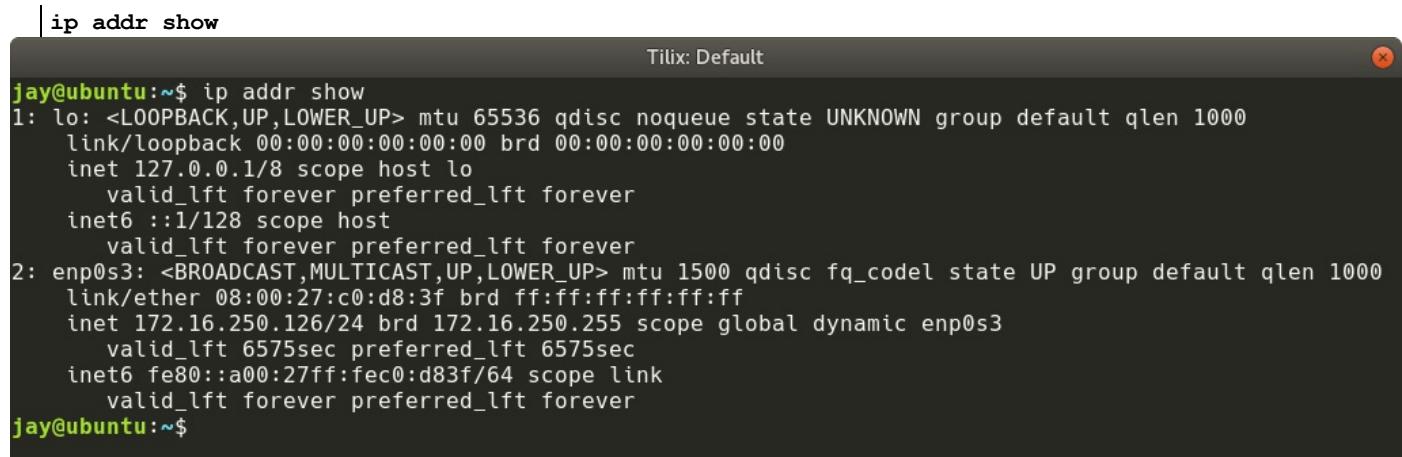
# The following lines are desirable for IPv6 capable hosts
::1      localhost ip6-localhost ip6-loopback
ff02::1  ip6-allnodes
ff02::2  ip6-allrouters
jay@ubuntu:~$ █
```

Sample contents from a /etc/hosts file

By using a text editor and editing this file, we can change this occurrence of the hostname. Considering the extra work of changing our hostname, I see little benefit in using `hostnamectl` over manually editing `/etc/hosts` and `/etc/hostname`, so the choice is pretty much yours. You'll end up using a text editor to update `/etc/hosts` anyway, so you may as well perform the update the same way for both.

Managing network interfaces

Assuming our server's hardware has been properly detected, we'll have one or more network interfaces available for us to use. We can view information regarding these 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
Tilix: Default
jay@ubuntu:~$ 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:c0:d8:3f brd ff:ff:ff:ff:ff:ff
    inet 172.16.250.126/24 brd 172.16.250.255 scope global dynamic enp0s3
        valid_lft 6575sec preferred_lft 6575sec
    inet6 fe80::a00:27ff:fec0:d83f/64 scope link
        valid_lft forever preferred_lft forever
jay@ubuntu:~$
```

Viewing IP information with the `ip addr show` command

If for some reason you're not fond of typing, you can shorten this 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 (remove its IP assignment and 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 18.04 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 say `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 single network card (`eth0`) and you add a second (which then becomes `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 one of my older installations:

```
| SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="01:22:4e:a5:f2:ec", ATTR{dev_id}=
```

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, `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. 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, offer it as an additional package you can install. In the case of Ubuntu Server 18.04, `net-tools` commands such as `ifconfig` are alive and well. However, this package may no longer be included by default sometime in the future, so if you find that you have to install it manually, you'll know why.

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):

```
| /sbin/ifconfig
```

The `/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`, use the fully qualified command instead:



```
jay@ubuntu:~$ /sbin/ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 172.16.250.126 netmask 255.255.255.0 broadcast 172.16.250.255
              inet6 fe80::a00:27ff:fe00:d83f prefixlen 64 scopeid 0x20<link>
                ether 08:00:27:c0:d8:3f txqueuelen 1000 (Ethernet)
                  RX packets 49607 bytes 71252851 (71.2 MB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 8166 bytes 737023 (737.0 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 152 bytes 11792 (11.7 KB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 152 bytes 11792 (11.7 KB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```
jay@ubuntu:~$
```

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 main 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.

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 get a permanent IP address in place right away. 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 stick. But in the case of virtual or physical servers you manage yourself, you'll start off with a dynamic address.

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 7](#), 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 is outside of the range that your DHCP server assigns. 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 obeys you and 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 like the former because it sounds cooler). 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, which I'll go over in [Chapter 7](#), Setting Up Network Services.

But you don't always have a choice. It's often the case, that as a Linux administrator, you may or may not be in charge of the DHCP server. At my organization, our DHCP server runs Windows, so I don't touch it. I let the Windows administrators deal with it. Therefore, you may be given a free IP address from your network administrator, and then you'll proceed to configure your Ubuntu Server to use it.

There are now multiple methods for setting an IP address manually on an Ubuntu Server, and which method you use depends on which version of Ubuntu you're running. Since Ubuntu 17.10 (the version directly preceding Ubuntu 18.04) the way in which you set a manual IP has completely changed. First, I'll go over the new method (using **Netplan**) and then I'll go over the method we can use in legacy versions of Ubuntu, in case you end up managing such a server.

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 on a server running Ubuntu 17.10 or newer, you should see at least one file there, `01-netcfg.yaml`. It's possible the file could be saved with a different name, such as `50-cloud-init.yaml`. On my server, the file looks like this:

```
# This file describes the network interfaces available on your system
# For more information, see netplan(5).
network:
  version: 2
  renderer: networkd
  ethernets:
    enp0s3:
      dhcp4: yes
```

We can already glean some obvious information from this default file. First, we know that the interface will utilize DHCP in order to grab an IP address, which we can tell from the last line. We can also tell that this configuration file is related to interface `enp0s3`. One thing that may not be immediately obvious is the `renderer` line, which is set to `networkd` in this case. This tells us that the service that is managing this interface is `systemd-networkd`, part of the `systemd` suite of tools. Many things on the system are managed by `systemd` (which is pretty much taking over everything nowadays). An alternative to `systemd-networkd` is `NetworkManager`, but we're not going to get into that just yet.

You can probably guess one edit to this file we need to make in order to assign an IP address manually. The last line, `dhcp4: yes` should be changed to `dhcp4: no`. Right underneath that, we'll add a new line:

```
| addresses: [192.168.0.101/24]
```

Of course, you'd add whatever IP address you want, I used a standard `/24` address as an example. However, we're also going to need a default gateway and DNS address, so we have more lines to add. Here is the complete file, with the new lines we need added to the end:

```
# This file describes the network interfaces available on your system
# For more information, see netplan(5).
network:
  version: 2
  renderer: networkd
  ethernets:
    enp0s3:
      dhcp4: no
      addresses: [192.168.0.101/24]
      gateway4: 192.168.1.1
      nameservers:
        addresses: [192.168.1.1,8.8.8.8]
```

In the sample file, I bolded configuration lines that were either changed or newly added. In a nutshell, I added an IP address of `192.168.0.101` with a CIDR mask of `/24`. (We'll get into the specifics of how IP addresses work in a later chapter). Next, I set the default gateway to `192.168.1.1`. Note that the option used was `gateway4`. If I was using a default gateway that had an IPv6 address, I would've needed to use the `gateway6` option instead. In fact, I could've also set an IPv6 address by adding it to the same line where I declared the IPv4 address, making it look like this:

```
| addresses: [192.168.0.101/24, '2002:2::4/64']
```

Finally, I set the DNS servers to `192.168.1.1` and `8.8.8.8`. This is to simulate a situation in which the server would resolve names from a local DNS server and an external one (I used Google's `8.8.8.8` as an example).

Next, we'll need to apply and test these changes. This is something you probably don't want to do over SSH, since as soon as you activate these changes your SSH connection will drop. If you are using a virtual machine, you may want to make the changes from the VM console. If you're updating a physical machine, you may want to have a display attached. 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 it will apply the changes if not. The new IP address will take effect immediately.

Next, let's take a look at the procedure as it was in legacy versions of Ubuntu (basically, any version of Ubuntu older than 17.10). To set a manual IP on such a server, we will need to edit the `/etc/network/interfaces` file. This file is a single place for you to manage settings for each of your network interfaces on older systems. An example of this file follows. I've left out the output regarding the loopback adapter, and what you'll see is the section relating to the primary network card on one of my older servers:

```
# The primary network interface
auto enp0s3
iface enp0s3 inet dhcp
```

With this default file, we have a primary network interface (`enp0s3`), and we're allowing this interface to receive a dynamic IP assignment from a DHCP server. You can see this on the third line, where `dhcp` is explicitly called. If you haven't changed your `/etc/network/interfaces` file, and you're not using a VPS, yours will look very similar to this, with the interface name being the main difference.

If we wanted to manually assign a static IP address, we would need to make some changes to this file. Here's an example interfaces file, configured with a static address:

```
# The primary network interface
auto enp0s3
iface enp0s3 inet static
    address 10.10.96.1
    netmask 255.255.255.0
    broadcast 10.10.96.255
    dns-search local.lan
    dns-nameservers 10.10.96.1
```

I've bolded the sections of the output that I've changed. Essentially, I've changed `dhcp` to `static` and then added five additional lines. With this configuration, I'm assigning a static IP address to `10.10.96.1`, setting the subnet mask to `255.255.255.0`, and the broadcast address to `10.10.96.255`. For name resolution, I'm setting the DNS search to `local.lan` (you can omit this line if you don't have a domain), and the DNS server address to `10.10.10.96.1`. If you wanted to configure a static IP address for your server, you would simply change this output to match your environment, making especially sure to change your interface name if it's not the same as mine (`enp0s3`), as well as the values for your connection.

Now that we've edited our `interfaces` file, we'll need to restart networking for the

changes to take effect. Like I mentioned before, you would need to take some precautions here before you do so. The reason for this is, if you're connected to the server via a remote connection (such as SSH), you will lose connection as soon as you restart networking. Even worse, the second half of the network restart (the part that actually brings your interfaces back online) won't execute because the restart of networking will drop your remote connection before it would've completed. This, of course, isn't an issue if you have physical access to your server. The restart command (which I'll give you shortly), will complete just fine in that case. But with remote connections, your networking will go down if you restart it. The command to restart networking is the following:

```
| sudo systemctl restart networking.service
```

On older Ubuntu Servers (before `systemd`), you would use the following command instead:

```
| sudo /etc/init.d/networking restart
```

At this point, your new IP address configuration should take effect.

In the case of utilizing remote connections while configuring networking, you can alleviate the issue of being dropped before networking comes back up 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 case because it keeps commands running in the background, even if our connection to the server gets dropped. This is useful regardless of whether or not your system has an `interfaces` file or uses Netplan. 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 enter `tmux` prior to executing one of the `restart` commands for your server. You'll still (probably) get dropped from your shell, but the command will complete in the background, which means that networking will go down, and then come back up with

your new configuration.



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.

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 addr show` or `ifconfig`. If, for some reason, you cannot reconnect to the server, you may have made a mistake while editing the `/etc/network/interfaces` file. In that case, you'll have to walk over to the console (if it's a physical server), or utilize your virtual machine manager to access the virtual console to log in and fix the problem. 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.

Understanding NetworkManager

NetworkManager is a utility for managing network connectivity on your server, though it's not for everyone, and has largely been replaced with Netplan. This utility comes installed by default with older versions of Ubuntu Server, but it may not be included even on older versions if you're using a cloud appliance. Therefore, feel free to skip this section if you're not using a legacy version of Ubuntu Server as this won't apply to you otherwise.

NetworkManager is a service (also known as **daemon**) that runs in the background and manages your network connections. On a desktop Linux distribution, it does everything from managing connectivity, to managing your wireless networks. With it, you can configure profiles, and create custom connections you can switch between.

On servers, the added features of NetworkManager on a desktop distribution aren't all that useful, but the network connectivity management portion certainly is. With NetworkManager, it keeps an eye on whether or not your server is connected to a network. It even goes as far as to launch your DHCP client when network connectivity is started or restored, to facilitate the fetching of a dynamic IP address. If your network connection drops for any reason, NetworkManager will request a dynamic address as soon as it comes back online.

If you're not using DHCP however, NetworkManager is of little benefit. The best rule of thumb in my opinion is to use NetworkManager when you're using DHCP for IP assignments on older servers and to disable it otherwise.

Stopping and disabling NetworkManager can be done with the following commands (which you should only do if you've set up a manual static IP assignment):

```
| sudo systemctl stop NetworkManager  
| sudo systemctl disable NetworkManager  
| sudo systemctl restart networking
```

As I've mentioned before, my preferred approach is static leases via DHCP (in other words, DHCP reservations for servers). The added benefit of static leases is that you benefit from being able to utilize NetworkManager. It will keep your connection alive, and any time it reaches out for an IP address, your server will always get the address you've set aside for it in your DHCP server. (Again, we'll go over how to set this up in [c](#))

([Chapter 7](#), Setting Up Network Services). In situations where you need to set up a static IP, you may as well just disable NetworkManager because it may just get in your way.

Regarding the desktop version of Ubuntu, NetworkManager is a great asset. It allows you to easily manage Wi-Fi networks, VPN connections, wired network profiles, and more in one easy to use graphical tool. For servers though, NetworkManager isn't as useful in most cases.

Understanding Linux name resolution

In [Chapter 7](#), 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 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 (`minecraftserver.local.lan`) 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 minecraftserver
```

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 doesn't scale. 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 server, 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.

On legacy Ubuntu servers, there was a file, `/etc/resolv.conf`, that was used to determine which DNS servers to check. This is not used anymore in Ubuntu 18.04, since name resolution is now handled by `systemd-resolved`. For the sake of completeness though, here is a brief overview of this file 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, `/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 network assignment on this server is actually being managed by NetworkManager, as you can see from the warning comment included in the file. You can certainly alter this file so that the server will check different DNS servers, but the next time its IP address renews, or the connection is refreshed, this file will be overwritten and will be reset to use whatever DNS servers your DHCP server tells it to use. Therefore, if you really want to control how your server resolves names you can create a static assignment, as we did earlier in this chapter when configuring Netplan. This isn't a problem if you're using a static IP assignment.

Now that newer Ubuntu servers utilize `systemd-resolved` to handle name resolution, how do you see what name servers your server is currently using? You could consult the Netplan configuration if you're utilizing a static IP address, but if your server received an IP address via DHCP, that file won't help you. The following command will let you know what DNS nameservers your server is currently pointing to:

```
| systemd-resolve --status |grep DNS\Servers
```

Tilix: Default

```
jay@dev:~$ systemctl-resolve --status |grep DNS\Servers
  DNS Servers: 172.16.250.1
jay@dev:~$
```

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 7](#), Setting Up Network Services, but you should now understand how the name resolution process works on your Ubuntu Server.

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 7](#), Setting Up Network Services, but this utility is very handy, and we should start using it as soon as possible. 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.

OpenSSH allows you to open a command shell on other Linux servers, allowing you to run commands as if you were there in front of the server. In a Linux administrator's workflow, they will constantly find themselves managing a plethora of machines in different locations. OpenSSH works by having a daemon running on the server that listens for connections. On your workstation, you'll use your SSH client to connect to the server, to begin running commands on it. SSH isn't just useful for servers either; you can manage workstations with it, as well as network appliances. I've even used SSH on my laptop to connect to my desktop to issue a `reboot` command, just because I was too lazy to walk all the way to my bedroom. It's extremely useful. And thankfully, it's also very easy to learn.

Depending on the choices you've made during installation, your Ubuntu Server probably has the OpenSSH server installed already. If you don't remember, just type `which sshd` at your shell prompt. If you have it installed, your output should read `/usr/sbin/sshd`. If you haven't installed the server, you can do so with the following command:

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

Keep in mind, though, that the OpenSSH server is not required to connect to other machines. Regardless of whether or not you install the server, you will still have the OpenSSH client installed by default. If you type `which ssh` at your shell prompt (omitting the `d` from `sshd`) you should see an output of `/usr/bin/ssh`. If, for some reason, you don't have this package installed and you received no output from this command (which would be rare), you can install the OpenSSH client with the following command:

```
| sudo apt install openssh-client
```

I want to underline the fact that you're not required to install the `openssh-server` package in

order to make connections to other machines. You only need the `openssh-client` package to do that. For the vast majority of Linux administrators, I cannot think of a good reason to not have the `openssh-client` package installed. It's useful when you want to remotely manage another Linux machine or server, but by itself, it doesn't allow other users to connect to you. It's important to understand that installing the OpenSSH server does increase your attack surface, though.

Whether or not to run an OpenSSH server on your machine usually comes down to a single question: "Do you want to allow users to connect remotely to your server?" Most of the time, the answer to that question is **yes**. The reason is that it's more convenient to manage all your Linux servers at your desk, without having to walk into your server room and plug in a display and keyboard every time you want to do something. But even though you most likely want to have an SSH server running, you should still keep in mind that OpenSSH would then be listening for and allowing connections. The fact that OpenSSH allows you to remotely manage other servers is its best convenience, and also its biggest weakness. If you are able to connect to a server via SSH, so can others.

There are several best practices for security that you'll want to implement if you have an OpenSSH server running. In [Chapter 15](#), 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 [Chapter 15](#), Securing Your Server, to read the section there that talks about securing OpenSSH, and then come back here when you're done. If you have a server that is directly accessible via the internet, with users with weak passwords and you are allowing connections via SSH, I can personally guarantee that it will be hijacked within two weeks. As a general rule of thumb though, you're usually fine as long as your user accounts have strong passwords, the OpenSSH package is kept up to date with the latest security updates, and you disable login via `root`.

With all of that out of the way, we can get started with actually using 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. To verify, run the following command:

```
| systemctl status ssh
```

```
Tilix: Default
jay@dev:~$ systemctl status ssh
● ssh.service - OpenBSD Secure Shell server
  Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset:
    Active: active (running) since Sat 2018-01-27 11:05:30 EST; 2h 31min ago
      Main PID: 709 (sshd)
        Tasks: 1 (limit: 4915)
       CGroup: /system.slice/ssh.service
               └─709 /usr/sbin/sshd -D

Jan 27 11:05:30 ubuntu systemd[1]: Starting OpenBSD Secure Shell server...
Jan 27 11:05:30 ubuntu sshd[709]: Server listening on 0.0.0.0 port 22.
Jan 27 11:05:30 ubuntu sshd[709]: Server listening on :: port 22.
Jan 27 11:05:30 ubuntu systemd[1]: Started OpenBSD Secure Shell server.
Jan 27 11:05:52 ubuntu sshd[867]: Accepted password for jay from 172.16.250.
Jan 27 11:05:52 ubuntu sshd[867]: pam_unix(sshd:session): session opened for
Jan 27 13:12:53 dev.mynetwork.org sshd[13884]: Accepted password for jay fro
Jan 27 13:12:53 dev.mynetwork.org sshd[13884]: pam_unix(sshd:session): sessi
lines 1-16/16 (END)
```

Output from systemctl, showing a running SSH server

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
```

On older Ubuntu servers (for example, 14.04 and 12.04), you can use the following two commands in order to start and enable the OpenSSH server respectively:

```
| sudo service ssh start
| sudo update-rc.d ssh defaults
```



Don't worry about the `systemctl` or `service` commands just yet, we'll go over them in greater detail in [Chapter 6](#), Controlling and Monitoring Processes.

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 netstat -tulpn |grep ssh
```

```
Tilix: Default
jay@dev:~$ sudo netstat -tulpn |grep ssh
tcp        0      0 0.0.0.0:22          0.0.0.0:*              LISTEN      709/sshd
tcp6       0      0 :::22              :::*                  LISTEN      709/sshd
jay@dev:~$
```

Output from netstat showing that 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, keep in mind that this file is the main 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 10.10.96.10
```

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@10.10.96.10
```

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@10.10.96.10
```

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 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 with SSH. When you're finished with your session, simply type `exit` at the shell prompt, or press `Ctrl + D` on your keyboard.



If you started background commands on the target via SSH, use Ctrl + D to end your session, otherwise those processes will be terminated. We'll talk about background processes in [Chapter 6](#), 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 7](#), Setting Up Network Services, with regards to securing it, though. In the next section, we'll discuss SSH key management, which also 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 benefit to this 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 an even larger margin. We'll go over that later in the book.

To get started, you'll first need to generate your key. To do so, use the `ssh-keygen` command as your normal user account. The following screenshot shows what this process generally looks like:

```
Tilix: Default
jay@dev:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/jay/.ssh/id_rsa):
Created directory '/home/jay/.ssh'.
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:BmKcsuw8cjw10aJA/6En9kjzDAP0oUyu07WFV2uTqF4 jay@dev.mynetwork.org
The key's randomart image is:
+---[RSA 2048]---+
| + .
| * +..o
| .=.oB.+
| .o.*o*+.+
| . *0*o= S
| *o*@o o
| + X.E+
| = +
| .
+---[SHA256]---+
jay@dev:~$
```

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 writable by the owner. You can see this yourself by executing `ls -l /home/<user>/.``ssh`:

```
Tilix: Default
jay@dev:~$ ls -l /home/jay/.ssh
total 8
-rw----- 1 jay jay 1675 Jan 27 13:51 id_rsa
-rw-r--r-- 1 jay jay 403 Jan 27 13:51 id_rsa.pub
jay@dev:~$
```

Listing the contents of the .ssh directory, showing the permissions of the newly-created keys

The public key is the key that gets copied to other servers to facilitate you being able to log in via the 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 then lets you log in. You'll also be asked for your passphrase, if you chose to set one.

To actually transmit your public key to a target server, we use the `ssh-copy-id` command. In the following example, I'll show a variation of the command that's copying the key to a server named `unicorn`:

```
| ssh-copy-id -i ~/.ssh/id_rsa.pub unicorn
```

With that command, replace `unicorn` with the hostname of the target server, or its IP address if you don't have a DNS record created for it. 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:

```
Tilix: Default
14:06:51 [0] [seraph:~] % ssh-copy-id -i ~/.ssh/id_rsa.pub unicorn
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/jay/.ssh/id_rsa.pub"
/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@unicorn.local.lan's password:

Number of key(s) added: 1

Now try logging into the machine, with:  "ssh 'unicorn'"
and check to make sure that only the key(s) you wanted were added.

14:07:10 [0] [seraph:~] %
```

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, an `.ssh` directory is created in your `home` directory if it didn't already exist. Inside that directory, a file named `authorized_keys` is created if it didn't exist. 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 via SSH, SSH checks the contents of the `~/.ssh/authorized_keys` file on that server, looking for a key that matches the private key (`~/.ssh/id_rsa`) on your machine. If the two keys are a mathematical 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 password-less authentication, meaning you won't be asked to enter anything when authenticating. Having a key relationship with your server is certainly more secure than authenticating to SSH via a password, but it's even more secure with a passphrase and creating one is definitely recommended. Thankfully, using a passphrase doesn't have to be inconvenient. 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.

To benefit from this, your SSH agent must be running. 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, that will continue to run in the background of your shell. Then, you can unlock your key for your agent to use:

```
| ssh-add ~/.ssh/id_rsa
```

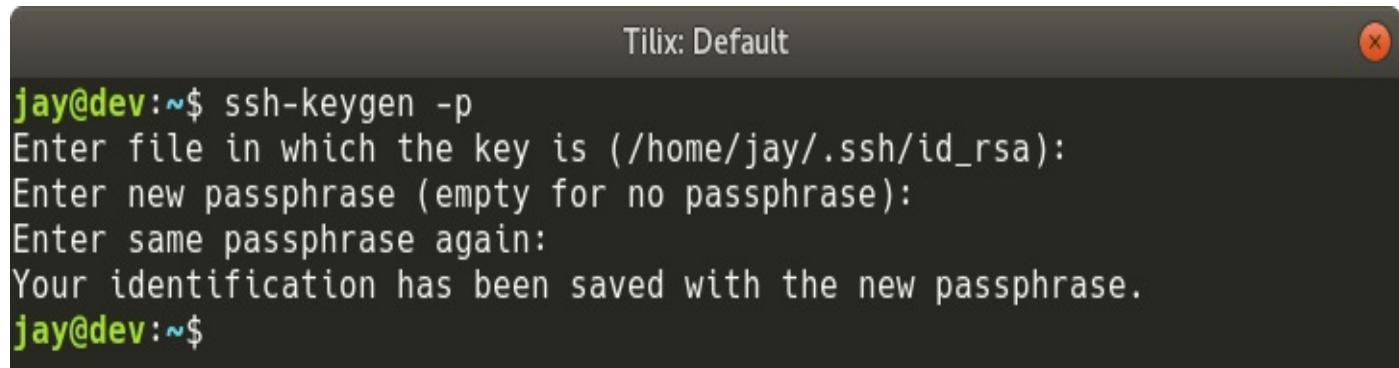
At this point, you'll be asked for your passphrase. As long as you enter it properly, your key will remain open and you won't need to enter it again for future connections, until you close that shell or log out. This way, you can benefit from the added security of having a passphrase, without the inconvenience of entering your passphrase over and

over.

There will, at some point, be a situation where you'll want to change your SSH passphrase. To do that, you can execute the following command, which will allow you to add a passphrase if you don't have one already, or change an existing passphrase:

```
| ssh-keygen -p
```

Once you enter the command, press Enter to accept the default file (`id_rsa`). 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:



```
Tilix: Default
jay@dev:~$ ssh-keygen -p
Enter file in which the key is (/home/jay/.ssh/id_rsa):
Enter new passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved with the new passphrase.
jay@dev:~$
```

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 SSH, 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 would look something like this:

```
| /home/jay/.ssh/config
```

This file doesn't exist by default, but if it's found, SSH will parse it 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.local.lan
  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.local.lan
```

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.local.lan`, 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 server `myserver`), 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 `potoato` 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 a look 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. While we'll revisit OpenSSH in [chapter 15](#), 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!

Questions

1. A server's _____ identifies it on the network.
2. Which command can be used to show current network connection information (such as the assigned IP address)?
3. _____ is a utility that allows you to define network configuration via YAML.
4. _____ allows you to connect to other Linux machines and run commands as if you had a keyboard and display attached to them.
5. Adding a _____ to your SSH key improves its security.
6. A _____ IP can be assigned to a server so that its IP address never changes.
7. Which file defines the order in which resources are checked for name resolution?

Further reading

- **Netplan FAQ:** <https://netplan.io/faq>
- **Ubuntu SSH key documentation:** <https://help.ubuntu.com/community/SSH/OpenSSH/Keys>

Managing Software Packages

Prior to this chapter, I've already walked you through several situations that required you to install a new package or two on your server, but we have yet to have any formal discussion on package management. In this chapter, we'll explore package management in more detail. The Ubuntu platform has a huge range of software available for it, featuring packages for everything from server administration to games. In fact, as of the time I'm writing this chapter, there are over 50,000 packages in Ubuntu's repositories. That's a lot of software and, in this chapter, we'll take a look at how to manage these packages. 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
- Taking advantage of hardware enablement updates
- 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 orphaned packages
- Making use of Aptitude

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 the Google Play store, Apple offers its App Store, and so on. For us Linux folk, 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, 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. Sometimes, the new version of a package is simply a security update. 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 maintainer will review the claim and then release an updated version to correct it. This process happens very quickly, as I've seen severe vulnerabilities patched even on the same day they were reported in some cases. The Ubuntu developers are definitely on top of their game in taking care of security issues.

Also, new versions of packages are sometimes a feature update, which is an update released to introduce new features and isn't 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 introduce instability. Instead, known working and stable packages are preferred, but given the fact that Ubuntu releases every six months, you don't have to wait very long.

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 15](#), 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 feature updates coming regularly. With just one command (which we'll get to shortly), you can install a package along with all of its dependencies. Having done manual dependency resolution myself, I can tell you first-hand that having dependencies handled automatically is a very 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.

Taking advantage of hardware enablement updates

One issue that's been the case in the Linux industry has been hardware support. This is problematic in various Linux distributions because you may find yourself in a situation where your server (or even a laptop) is 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 LTS releases. We discussed the difference between LTS and regular releases back in [Chapter 1](#), Deploying Ubuntu Server, and HWE updates are exclusive to LTS. HWE updates are optional, and will include a new kernel, and typically additional updated software to handle the latest video cards. This way, you can be sure your newer hardware will still be supported by the current Ubuntu LTS release. The great thing about this is that HWE updates allow you to stay on an LTS release, while still benefiting from the newer drivers of non-LTS releases.

The way in which an administrator acquires these updates has changed a bit from previous releases. In the past, you would download and install a newer version of your Ubuntu LTS release, which would have the newer software built right in. While you can still do this, what's different is that you can now upgrade to the latest HWE stack as soon as it's released, and have it included when you install the latest package updates.

It's important to note that whether or not you receive HWE updates is a choice you have to make when it comes to the server version of Ubuntu. 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 currently, 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 18.04 is new to the scene so HWE updates haven't been released yet, and the method may be subject to change. With Ubuntu 16.04, HWE updates weren't released until Ubuntu 16.04.2, as generally these updates aren't released until the second point release. Assuming Ubuntu 18.04 follows the same plan, you'll most likely see an option such as the one in the following screenshot once 18.04 is old enough to receive this feature:



Main menu of the Ubuntu installer

If you've already installed Ubuntu Server, you can install the latest HWE stack from the Terminal. In Ubuntu 16.04 for example, you can switch to the HWE kernel with the following command:

```
| sudo apt install --install-recommends linux-generic-hwe-16.04
```

When Ubuntu 18.04 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.

Understanding the differences between Debian and Snap packages

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

Traditionally, each distribution has their own package format, and their own utilities to manage them. Nowadays, there's a push to adopt a single package format that each distribution can install. Contenders for this single package format include Flatpak, AppImage, and Snap packages. Specific to Ubuntu, it 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).

Debian packages are basically what you've been installing throughout the book. At no point (yet) have we installed anything else, unless you've read ahead. 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.

Debian packages present some challenges and major cons, though. First, the entire distribution is made up of Debian packages. This means that the Linux kernel, system packages, libraries, and security updates are all Debian packages. When you install Ubuntu Server, it installs these packages during the process. When you install security updates, Debian packages are being 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 the system packages. Hypothetically, 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 issue with Debian packages is software availability. When a new 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. The developers can make exceptions, but they usually don't. If you ask me, having to wait for the next release of your operating system in order to have a newer version of your favorite software is just silly. After all, you don't have to install a new version of Windows or macOS just to install newer applications!

Canonical, the makers of Ubuntu, understand these pain points and have been making a great effort to change how packages are managed. In fact, they actually came up with an entirely new type of package—the **Snap package**. Snap packages (or more simply, Snaps) have no impact on the underlying Debian packages at all and are completely separate. This is great, because there will not be a situation in which you update a Snap package and it conflicts with your system packages. This also allows you to have a newer version of an application than what would otherwise be made available. Since Snaps are separate 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 it requires 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? Normally, I would definitely lean toward Snaps, but you don't always have a choice. The reason being, Snaps are a new concept and not all developers have migrated their software to Snaps just yet. The majority of the packages you'll install throughout this book are Debian packages, mainly because that's just what's available right now. As time goes on, it's expected that the number of applications available in the Snap format will grow. But as a rule of thumb, try out the Snap version first, and fall back to the standard Debian packages if you need to or if what you're trying to install isn't available in Snap format yet.

Installing and removing software

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.

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 available (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. We've been using this tool for a while, but now we'll actually have some formal discussion about it. The `apt install` command is just one tool in a set of tools known as **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.

As you've seen throughout the book, we can use `apt` along with the `install` keyword and then a package name in order to install a new package on your system. 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 Debian package onto a Ubuntu system? 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 in which 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 see the following output on my system when I start to install it:

```
| sudo apt-get install apache2
|                                          Tilix: Default
|
jay@ubuntu:~$ sudo apt-get install apache2
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 ssl-cert
Suggested packages:
  www-browser apache2-doc apache2-suexec-pristine | apache2-suexec-custom openssl-blacklist
The following NEW packages will be installed:
  apache2 apache2-bin apache2-data apache2-utils libapr1 libaprutil1 libaprutil1-dbd-sqlite3
  libaprutil1-ldap ssl-cert
0 upgraded, 9 newly installed, 0 to remove and 0 not upgraded.
Need to get 1,619 kB of archives.
After this operation, 6,570 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

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 `www-browser`, `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.

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 15](#), 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.

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. This may seem like a no-brainer, but not everyone prefers this automation. As I've mentioned, the more packages installed on your server, the higher the attack surface, but running daemons 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 up 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 either be the package isn't found or `apt` bails out of the process when it encounters a package it's looking for but cannot find.

Removing packages is also very easy and follows a very similar syntax; you would only

need to replace the keyword `install` with `remove`:

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

And, just like with the `install` option, you can remove multiple packages at the same time as well. 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. 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`):

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

So, that concludes the basics of managing Debian packages with `apt`. Now, let's move on to managing Snaps. 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 to display a list of Snap packages available to us:

```
| snap find <package>
```

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
```

Now, if we check the location of `nmap` with the `which nmap` command, we see that the Snap version of `nmap` is run from a special place:

```
| which nmap
```

```
22:23:24 [0] [seraph:~] % which nmap
/snap/bin/nmap
22:24:30 [0] [seraph:~] %

```

Checking the location of the nmap binary

Now, when we run `nmap`, we're actually running it from `/snap/bin/nmap`. If we also have the `nmap` utility installed from Ubuntu's APT repositories, then we can run either one at any time, since Snap packages are independent of the APT packages. If we wish to run Ubuntu's version, we simply run it from `/usr/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:

```
| sudo snap refresh nmap
```

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. Unfortunately, there aren't very many packages available for us to work with just yet, but, as the technology advances and evolves, I'm sure many great applications will be available via this method. By the time this book is released, I'm sure the number of Snap packages will have already grown.

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 for even 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 already, 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 then 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 with the `apt show` command:

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

```
Tilix: Default
jay@ubuntu:~$ apt-cache show libapache2-mod-php
Package: libapache2-mod-php
Architecture: all
Version: 1:7.1+54ubuntu1
Priority: optional
Section: php
Source: php-defaults (54ubuntu1)
Origin: Ubuntu
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer: Debian PHP Maintainers <pkg-php-maint@lists.alioth.debian.org>
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Installed-Size: 16
Depends: libapache2-mod-php7.1
Filename: pool/main/p/php-defaults/libapache2-mod-php_7.1+54ubuntu1_all.deb
Size: 3170
MD5sum: 711cc5fe213ca88ebfdffa3b13d8e777
SHA1: 7d3e31eb369c95c03a31b6c0cdde99d2e7419803
SHA256: 1c929ec1c5e7cd3cfb28c8b0d51445f79788f2d1f402c571ec85224be6ef884
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 Ubuntu's default
  PHP version (currently 7.1).
Description-md5: 8727825cd3d8cdd7cf66a486def35e8d
Task: lamp-server
Supported: 5y
```

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 Debian 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-cache search` command, as well as the `snap find` commands, 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 and 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.

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 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 wide open for 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.

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/ bionic 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 which `apt` will use in order to reach the repository. In the third section, we have the code-name of the release; in this case, it's `bionic` (which refers to the code-name for Ubuntu 18.04, *Bionic Beaver*).

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 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.

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 or 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. In a perfect world, the application would be available as a Snap package, which would be the safest way to install it. PPAs are a way to obtain software outside of the default repositories in a situation where a Snap isn't available.

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>.

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 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.

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. 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
```

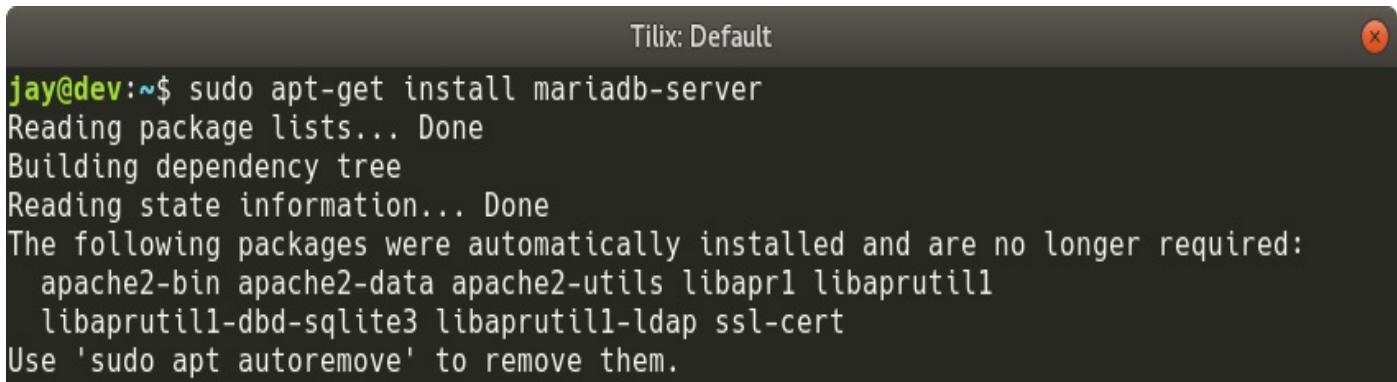
Normally, we simply use `apt` instead of `apt-get` nowadays, but some commands, such as `dselect-upgrade`, only work with `apt-get`.

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.

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 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@dev:~$ sudo apt-get install mariadb-server
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  apache2-bin apache2-data apache2-utils libapr1 libaprutil1
  libaprutil1-dbd-sqlite3 libaprutil1-ldap ssl-cert
Use 'sudo apt autoremove' to remove them.
```

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 `mariadb-server`. 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.

Although we haven't walked through updating packages (we'll do that in a later chapter),

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. 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. Later in the next section, I'll show you how to inform APT that you'd rather keep a package that is marked as an orphan, in case you run into a situation where APT lists a package for cleanup that you'd rather keep.

Making use of Aptitude

The `aptitude` command is a very useful, text-based utility for managing packages on your server. Some administrators use it as an alternative to `apt`, and it even has additional features that you won't find anywhere else. To get started, you'll need to install `aptitude`, if it isn't already:

```
| sudo apt install aptitude
```

The most basic usage of `aptitude` allows you to perform functions you would normally be able to perform with `apt`. In the following table, I outline several example `aptitude` commands, as well as their `apt` equivalent (each one assumes `sudo` since you need `root` privileges to modify packages):

aptitude command	apt equivalent
<code>aptitude install <packagename></code>	<code>apt install <packagename></code>
<code>aptitude remove <packagename></code>	<code>apt remove <packagename></code>
<code>aptitude search <search term></code>	<code>apt-cache search <packagename></code>
<code>aptitude update</code>	<code>apt update</code>
<code>aptitude upgrade</code>	<code>apt upgrade</code>
<code>aptitude dist-upgrade</code>	<code>apt dist-upgrade</code>

For the most part, the commands I listed in the preceding table operate in much the same way as their `apt` equivalents. The output from an `aptitude` command will typically look very close to the `apt` version. There are some noteworthy differences, though.

First, compare the output from the search commands. Both commands will list package names, as well as a description for each package. The `aptitude` version places additional blank space in between the application name and description, which makes it look nicer. However, it also includes an additional column that the `apt` version doesn't include, which represents the state of the package. This column is the first column you'll see when you search for packages with `aptitude`, and will show `i` if the package is already installed, `p` if the package is not installed, and `v` if the package is virtual (a virtual package exists merely as a pointer to other packages). There are other values for this column; feel free to consult the man page for `aptitude` for more information.

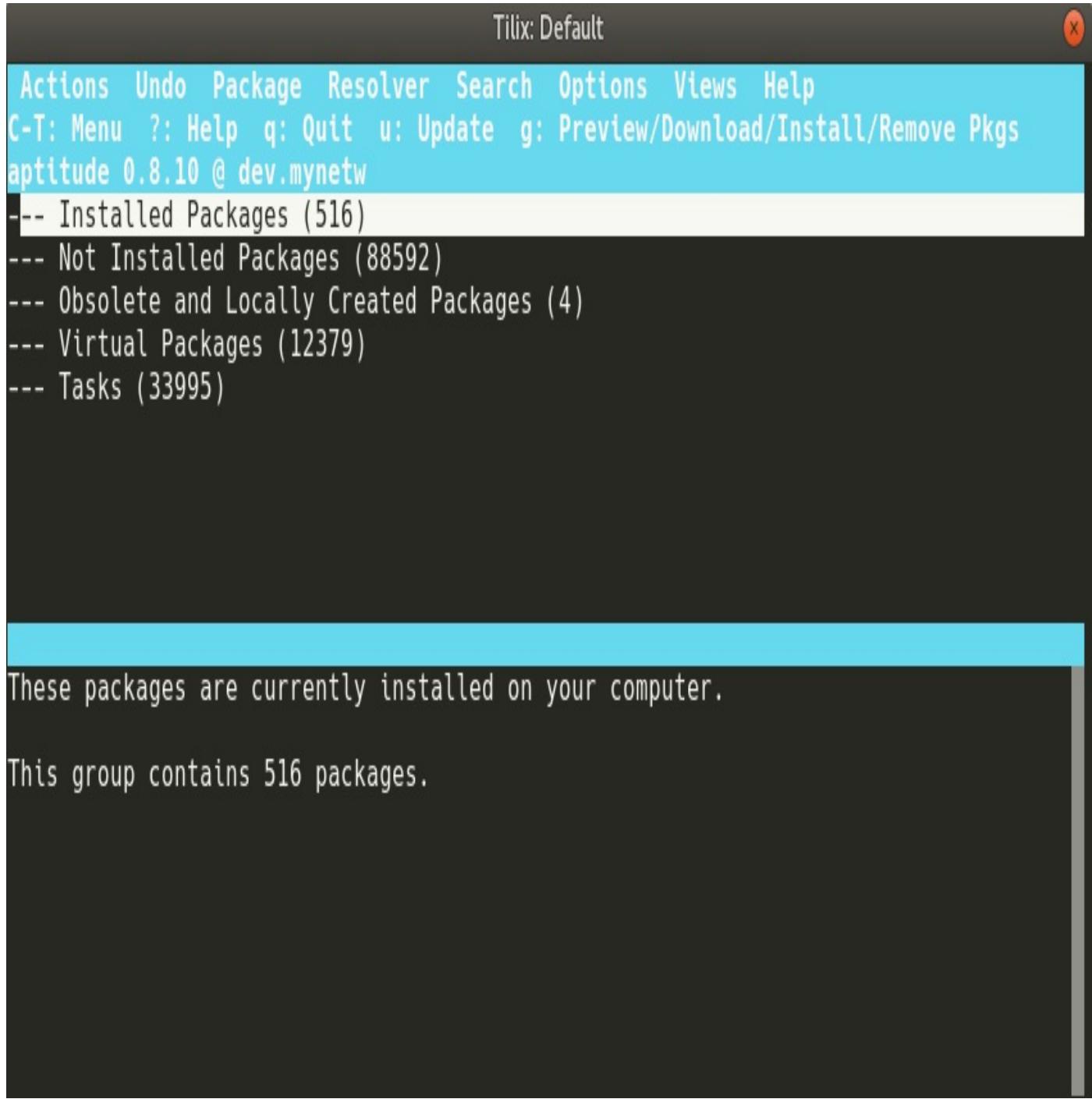
Another useful trick with `aptitude` is the ability to fix a situation where the `apt` command lists a package as no longer being necessary, even though you would rather keep it around. If you recall, we went over this subject a few sections ago. Basically, if a package is installed as a dependency for another package (but the original package was removed), the package will be marked as automatically installed, which means it becomes a candidate for cleanup with the `apt autoremove` command. In some cases, you may wish to keep such a package around, and you'd rather the `autoremove` command not remove it from your system. To unmark a package as automatically installed, `aptitude` comes to our rescue with the following command:

```
| sudo aptitude unmarkauto <packagename>
```

After you unmark a package as automatically installed, it will no longer be a candidate for `autoremove`, and you can then use the `apt autoremove` command without fear that the package will be removed.

However, the `aptitude` command offers yet another nice feature in the form of a (somewhat) graphical interface you can interact with on the shell, if you enter the `aptitude` command (as `root` or with `sudo`) with no arguments or options:

```
| sudo aptitude
```



Aptitude in action

The `aptitude` command features an `ncurses` interface, which is essentially a Terminal equivalent of a graphical application. You are able to interact with `aptitude` in this mode by using your arrow keys to move around, Enter to confirm selections, and q to quit. Using `aptitude`'s graphical utility, you are basically able to browse the available packages, which are split into several categories. For example, you can expand `Not Installed Packages` to narrow down the list to anything not currently installed, then you can narrow down the list further by expanding a category (you simply press Enter to expand a list). Once you've chosen a category, you'll see a list of packages within that category

that you can scroll through with your arrow keys. To manipulate packages, you'll access the menu by pressing `Ctrl + T` on your keyboard, which will allow you to perform additional actions. Once in the menu, you can mark the package as a candidate for installation by using your arrow keys to navigate to `Package`, and then you can mark the package as needing to be installed by selecting `Install`. You can also mark a package as needing to be removed or purged (meaning the package and its configuration are both removed). In addition, you can mark a package as automatically installed (`Mark Auto`) or manually installed (`Mark Manual`). Once you've made your selections, the first menu item (`Actions`) has an option to `Install/Remove` packages, which can be used to finalize your new selections or removals. To exit from `aptitude`, access the menu and then select `Quit`.



There's more that you can do with the graphical version of `aptitude`, so feel free to play around with it on a test server and read its man pages to learn more. In addition to being an awesome way to manage packages, it also has a built-in game of Minesweeper! Have a look around the application and see if you can find it.

As you can see, `aptitude` is a very useful utility, and some even prefer it to plain `apt`. Personally, I still use `apt`, though I always make sure `aptitude` is installed on my servers in case I need to benefit from its added functionality.

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 we can use to manage them. We began the chapter with a discussion on 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. The `aptitude` command is also a neat alternative to the `apt` suite of commands, and in this chapter, we looked at its GUI mode as well as how it differs from APT. Snap packages were also covered, which is an exciting up-and-coming technology that will greatly enhance software distribution on Ubuntu.

In [Chapter 6](#), Controlling and Monitoring Processes, we're going to take a look at monitoring and managing the processes running on our server. We'll take a look at job management, as well as starting and stopping processes.

Questions

1. The _____ command suite allows you to manage Debian packages on your server.
2. What are hardware enablement updates, and how do they benefit you?
3. What are two types of packages you can install in Ubuntu?
4. What command would you use to install a Debian package?
5. What command would you use to remove a Debian package?
6. What does the `snap refresh` command do?
7. The _____ command is an alternative to `apt` and even has a graphical user interface.
8. What is a package repository?
9. What is a PPA?
10. What is the command you would use to back up a list of currently installed Debian packages?
11. Which command would you use to clean orphaned packages?
12. What are some of the benefits of Snap packages?

Controlling and Monitoring Processes

On a typical Linux server, there can be over a hundred processes running at any given time. The purposes of these processes ranges from system services such as NTP 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.

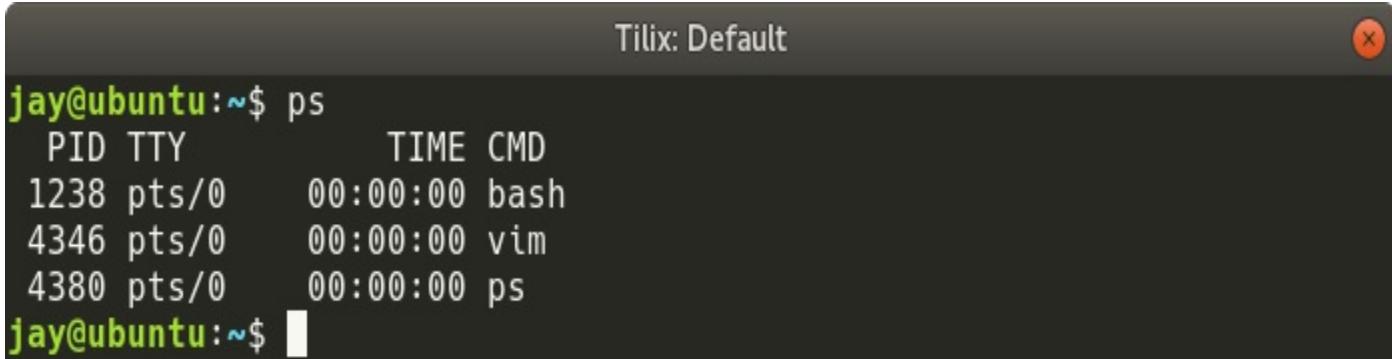
As we work through these concepts, we will cover:

- Displaying running processes with the `ps` command
- Managing jobs
- Dealing with misbehaving processes
- Utilizing `htop`
- Managing system processes
- Enabling services to run at boot time
- Managing processes on legacy servers with upstart
- Monitoring memory usage
- Scheduling tasks with cron
- Understanding load average

Showing running processes with the ps command

While managing our server, we'll need to understand what processes are running and how to manage these processes. 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.

When executed by itself, the `ps` command will show a list of processes run by the user that called the command:



```
jay@ubuntu:~$ ps
 PID TTY      TIME CMD
 1238 pts/0    00:00:00 bash
 4346 pts/0    00:00:00 vim
 4380 pts/0    00:00:00 ps
jay@ubuntu:~$
```

The output of the `ps` command, when run as a normal user and with no options

In the example screenshot I provided, 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 is what I'll refer to it as from now on. 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 involved 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 `4346`. You might run the following command:

```
| pidof vim
```



The output will give you the PID(s) of the process without you having to use the `ps` command.

Continuing with the `ps` command, there are several useful options you can give the command in order to change the way in which it shows its output. If you use the `a` option, you'll see more information than you normally would:

```
| ps a
```

PID	TTY	STAT	TIME	COMMAND
757	tty1	Ss+	0:00	/sbin/agetty -o -p -- \u --noclear tty1 linux
1238	pts/0	Ss	0:00	bash
4346	pts/0	T	0:00	vim
4386	pts/0	T	0:00	vim testfile
4387	pts/0	R+	0:00	ps a

```
jay@ubuntu:~$
```

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 the `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.

The `TTY`, `STAT`, and `TIME` fields are new; we didn't see those when we ran `ps` by itself. We

saw the other 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).

What may not be obvious at first are the `STAT`, `TIME`, and `TTY` fields. 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. Interruptible sleep means that the program is idle: it's waiting for input in order to awaken. 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, for 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.

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 which 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, processes run on a TTY, which is (for all intents and purposes) a Terminal that grabs input and manages that output, similar to a Teletypewriter in a virtual sense. A Terminal is our method of interacting with our server.

In the screenshot I provided, we have one process 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 environment 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 for its `TIME`. 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 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
Tilix: Default
jay@ubuntu:~$ ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START  TIME COMMAND
root      757  0.0  0.3  16116  1584 ttys000  Ss+  07:24  0:00 /sbin/agetty
jay       1238  0.0  1.0  22700  5340 pts/0    Ss+  07:25  0:00 bash
jay       4346  0.0  2.1  74600 10696 pts/0    T     08:24  0:00 vim
jay       4386  0.0  2.2  74804 10960 pts/0    T     08:34  0:00 vim testfile
jay       4394  0.0  0.6  41964  3440 pts/0    R+   08:42  0:00 ps au
jay@ubuntu:~$
```

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
```

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 are using the most CPU. 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.

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. Actually, we don't have to do that, unless we want to. 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.

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 using a text editor. 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:

```
| sudo apt install vim-nox
```



If you want to use `vim`, 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 want. Teaching you how to use a text editor is beyond the scope of this book, so just use whatever you feel comfortable with. In the following examples, I'll be using `nano`. However, 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 were returned to the shell so you could continue to use it. 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 `1070`:

```
| jay 1070 0.0 0.9 39092 7320 pts/0 T 15:53 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 background. Here's some example output:

```
| [1]- Stopped nano file1.txt
| [2]+ Stopped nano file2.txt
```

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. Better yet, I can 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'll get to the `htop` command later in this chapter, 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 back-grounded is its job ID and PID. 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 `vim` sessions, 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 quietly running in the background, allowing us to perform other tasks and allowing us to 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.

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 option 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` (zombie) processes in a situation where they don't close on their own. These processes are basically dead already and are typically waiting on their parent process 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 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. 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.

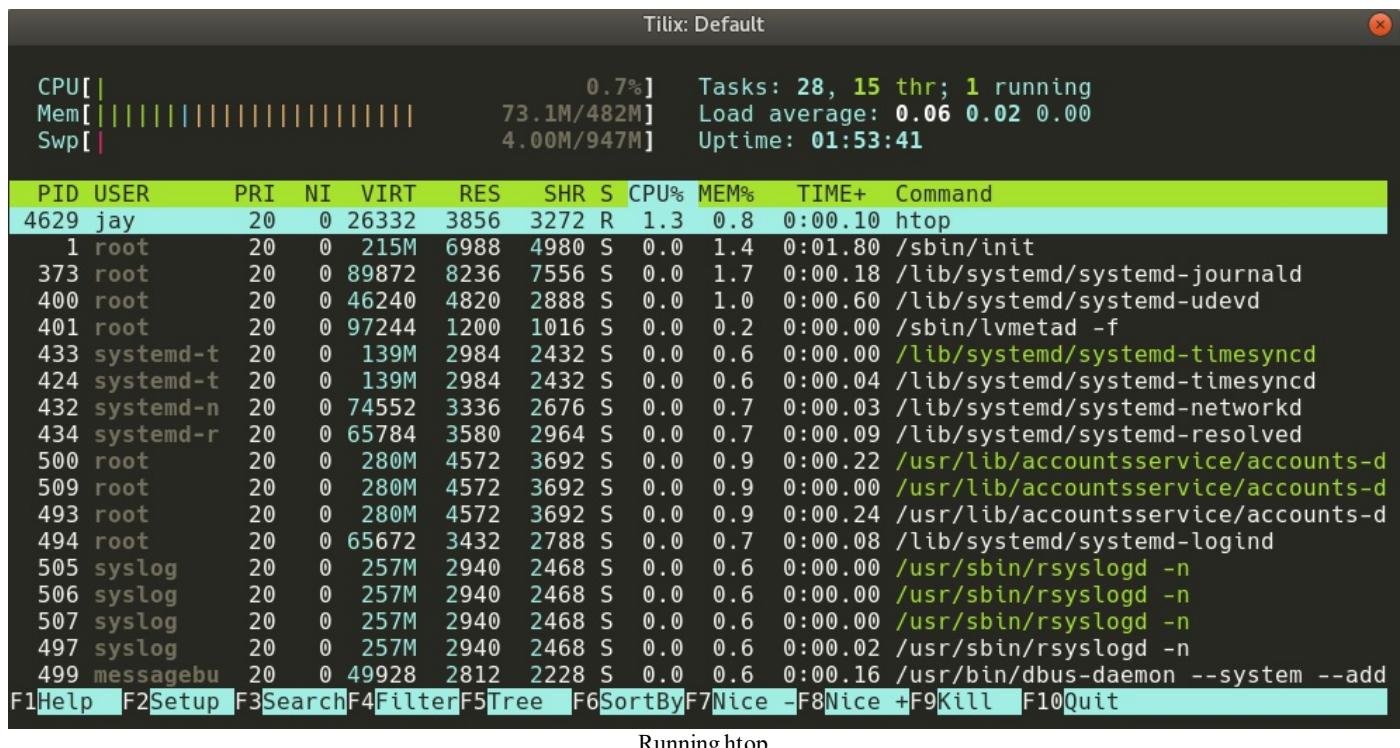
The `kill` and `killall` commands can be incredibly useful in the situation of a stuck process, but these are commands you would hope you don't have to use very often. Stuck processes can occur in situations where applications encounter a situation from which they can't recover, so if you constantly 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.

Utilizing 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, 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:



Running htop

At the top of the `htop` display, you'll see a progress meter for each of your cores (the server in my screenshot only has one core), as well as a meter for memory and swap. In addition, the upper portion will also show you 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 PID. 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 two 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.

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 respawn, 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.

As I've mentioned, `htop` updates its stats every two 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 seven 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.

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 the past several 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 in Ubuntu 18.04. Even though Ubuntu 18.04 uses `systemd`, and that is the preferred `init` system going forward, Ubuntu 14.04 is still supported at the time of writing, and it's very possible that you may have a mix of both as the older systems age out. For that reason, I'll go over both `init` systems in this section.

First, let's take a look at the way in which you manage services in modern Ubuntu releases with `systemd`. With `systemd`, services are known as **units**, though, for all intents and purposes, the terms mean the same thing. The `systemd` suite features the `systemctl` command, which allows you to start, stop, and view the status of units on your server. To help illustrate this, I'll use OpenSSH as an example, though you'll use the same command if you're managing other services, as all you would need to do is change the unit name.

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 a service, 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
● ssh.service - OpenBSD Secure Shell server
  Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
  Active: active (running) since Sun 2018-01-28 07:24:52 EST; 2h 4min ago
    Main PID: 724 (sshd)
      Tasks: 1 (limit: 4915)
     CGroup: /system.slice/ssh.service
             └─724 /usr/sbin/sshd -D

Jan 28 07:24:52 dev.mynetwork.org systemd[1]: Starting OpenBSD Secure Shell server...
Jan 28 07:24:52 dev.mynetwork.org sshd[724]: Server listening on 0.0.0.0 port 22.
Jan 28 07:24:52 dev.mynetwork.org sshd[724]: Server listening on :: port 22.
Jan 28 07:24:52 dev.mynetwork.org systemd[1]: Started OpenBSD Secure Shell server.
Jan 28 07:24:59 dev.mynetwork.org sshd[1100]: Accepted password for jay from 172.16.250.2 port 348
Jan 28 07:24:59 dev.mynetwork.org sshd[1100]: pam_unix(sshd:session): session opened for user jay
lines 1-14/14 (END)
```

Checking the status of a unit with `systemctl`

In my screenshot, you can see that I used `sudo` in order to run the `systemctl status ssh` command, but I didn't have to. You can actually check the status of most units without needing `root` access, but you may not see all the information available. Some units do not show the recent log entries without `sudo`. The `ssh` unit shows the same output either way, though.

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. As I've mentioned before, most packages in Ubuntu that include a service will enable it automatically, but other distributions which feature `systemd` may not. 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. A unit automatically becoming enabled and starting automatically may vary in packages that are configured differently, so make sure you check this output whenever you install a new application.

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. Not all units feature a `reload` option, so you should check the documentation of the application that provides the unit to be sure.

If a unit is not currently 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
```

The `systemd` suite includes other commands as well, such as `journalctl`. The reason I refer to `systemd` as a suite (and not just as an `init` system) is because it handles more than just starting and stopping processes, with more features being added every year. The `journalctl` command allows us to view log files, but I'll wait until [Chapter 16, Troubleshooting Ubuntu Servers](#), which is where I'll give you an introduction on how that works. 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.

If you have a server with an older version of Ubuntu Server installed, commands from the `systemd` suite will not work, as `systemd` is still relatively new. Older versions of Ubuntu use `upstart`, which we will take a look at now. If you don't manage any older Ubuntu

installations, feel free to skip the rest of this section.

Managing running services with `Upstart` is done with the `service` command. To check the status of a process with `Upstart`, we can do the following:

```
| service ssh status
```

In my case, I get the following output:

```
| ssh start/running, process 907
```

As you can see, we don't get as much output as we do with `systemctl` on `systemd` systems, but it at least tells us that the process is running and provides us with its PID. Stopping the process and starting it again gives us similar (limited) output:

```
| sudo service ssh stop
| ssh stop/waiting
| sudo service ssh start
| ssh start/running, process 1304
```

While the `Upstart` method of managing services may not give us as verbose information as `systemd`, it's fairly straightforward. We can start, stop, restart, reload, and check the status of a service of a given name. For the purposes of managing processes, that's pretty much it when it comes to `Upstart`.



If you're curious, `Upstart` stores the configuration for its services in the `/etc/init/` directory, with each service having its own file with a `.config` extension. Feel free to open one of these files to see how `Upstart` services are configured.

Some services are started using `init` scripts stored in `/etc/init.d`, rather than with `Upstart` or `systemd` commands. The `/etc/init.d` directory is used primarily with another `init` system, `sysvinit`, which is very old now by today's standards. However, some distributions that are still supported feature this even older `init` system, such as Debian 7 and Gentoo to name but two. In the early days, Ubuntu used `sysvinit` as well, before eventually replacing it with `Upstart`, so some of these `init` scripts remain, while some applications that have yet to be updated for newer `init` systems still use them. In most cases, `Upstart` and `systemd` can still manage these processes, but will call their `init` scripts in `/etc/init.d` if there is no built-in `Upstart` or `systemd` service file. Even if there is, you are still able to utilize this older method even on newer systems, but there's no telling how long this compatibility layer will remain. At the time of writing, though, the legacy `init` scripts are still well supported.

The older methods of starting, stopping, restarting, reloading, and checking the status of

services with the older `sysvinit` style is performed with the following commands respectively (again, using `ssh` as an example):

```
| /etc/init.d/ssh start  
| /etc/init.d/ssh stop  
| /etc/init.d/ssh restart  
| /etc/init.d/ssh reload  
| /etc/init.d/ssh status
```

At this point, with multiple `init` systems, it may be confusing as to which methods you should be using. But it breaks down simply. If you're using a version of Ubuntu that's 16.04 or newer, go with the `systemd` examples, since that is the new `init` system in use nowadays. If your server is using an Ubuntu version older than 16.04, use the `upstart` commands. In either case, if the normal commands don't work for a particular service, try the older `sysvinit` commands. 99.9% of the time, though, the `init` system that your version of Ubuntu Server ships with will work since just about any application worth using has been ported over.

The `systemd` init system actually has additional features that its predecessors didn't have. For example, you can enable a service to start as a particular user, rather than system-wide. To do so, the application has to support being started as a user, which would allow you to enable it with a command similar to the following:

```
| sudo systemctl enable myservice@myuser
```

In the hypothetical example I just provided, I'm enabling a unit called `myservice` for user `myuser`. A real-life example of why enabling a service to run as a user is useful is **syncthing**. Although an in-depth walk-through of syncthing is beyond the scope of this book, it's a great example since it supports being enabled system-wide as well as per user. This is a great thing, since syncthing is a file synchronization application, and each user on a system will have different files to sync, so each user can have his or her own configuration that won't affect other users. As usual, check the documentation that comes with the application you want to install to see what it supports as far as how to enable it.

Monitoring memory usage

Understanding how Linux manages memory can actually be a somewhat complex topic, as understanding how much memory is truly free can be a small hurdle for newcomers to overcome. You'll soon see that how Linux manages memory on your server is actually fairly straightforward. 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. 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. Giving the `free` command no option will result in the output being shown in terms of kilobytes:

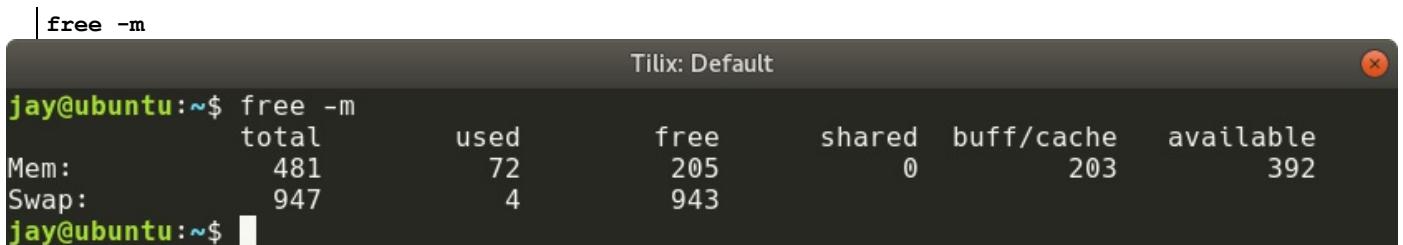


The screenshot shows a terminal window titled "Tilix: Default". The command entered is `free`. The output is as follows:

	total	used	free	shared	buff/cache	available
Mem:	493304	74452	210764	484	208088	402224
Swap:	969960	4096	965864			

Output of the `free` command

In my opinion, adding the `-m` option makes the `free` command much more readable:



The screenshot shows a terminal window titled "Tilix: Default". The command entered is `free -m`. The output is as follows:

	total	used	free	shared	buff/cache	available
Mem:	481	72	205	0	203	392
Swap:	947	4	943			

Output of the `free -m` command

Since everything is broken down into megabytes, it's much easier to read, at least for me.

To follow along, refer to the previous screenshot, and I will explain how to interpret the results of the `free` command.

At first glance, it may appear as though this server has only 205 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 392 MB in this case. That's now

much memory is actually free. Since this server has 481 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 205 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 if it needed it.

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. 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 these columns, I'll outline the meaning of each in the following table:

Column	Meaning
<code>total</code>	The total amount of RAM installed on the server.

used	Memory that is used (from any source). This is calculated as follows: total - free - buffers - cache.
free	Memory not being used by anything, cache or otherwise.
shared	Memory used by <code>tmpfs</code> .
buff/cache	The amount of memory being used by the buffers and cache.
available	Memory that is free for applications to use.

How much Swap is being used is something you should definitely keep an eye on. When 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.

Scheduling tasks with cron

Earlier in this chapter, we worked through starting processes and enabling them to run all the time and as soon as the server boots. In some cases, you may need an application to perform a job at a specific time, rather than always be 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 his or her own set of cron jobs (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 that executed the command. If you execute it as `root`, you'll see root'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` of other users:

```
| 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 the `crontabs` 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:

```
Tilix: Default
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. /usr/bin/vim.basic
 3. /usr/bin/vim.tiny
 4. /bin/ed

Choose 1-4 [1]:
```

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 will work:

```
| EDITOR=vim crontab -e
```

In that 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-get 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 Friday. We know this because the minute column is set to `*`, the hour column is set to `0`, the day column is `4` (Friday), and the command column shows a fully qualified command of `/usr/local/bin/cleanup.sh`.

A command being fully qualified is important with cron. For example, in the second example, we could have simply typed `apt-get update` for the command and that would probably have worked just fine. However, not including the full path to the program is considered bad cron etiquette. While the command would probably execute just fine, it depends on the application being found in the PATH of the user that's 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-get 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 am.

With the third example, we're running the `/usr/local/bin/run_report.sh` script on the first day of every month at 1:00 am. 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 1:00 am, 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.

Understanding load average

Before we close out this chapter, a very important topic to understand when monitoring processes and 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` or `top` command, the load average is shown within the output of each. 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 on 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 on the CPU at any one moment. 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 percent of usage to a low percent 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 if 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, are 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 and the more flexibility you have with the 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. 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. Each cashier is only able to handle one customer at a time. If there are more customers waiting to check out than 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).

Just like the cashier's, a 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 this 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. Going back to our supermarket example, this is 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.

It's normal for a server to always have a workload (so long as it's lower than the number of CPUs available), since that just means that our server is being utilized, which is the whole point of having a server to begin with (servers exist to do work). While typically, the lower the load average the better, depending on the context, it might actually be a cause for alarm if the load is too low. If your server's load average drops to an average of zero-something, 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 1x range that suddenly drops to 0x, 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.

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.

Summary

In this chapter, we learned how to manage processes and monitor our server's resource usage. 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 took a look at `htop`, which is a very handy utility for viewing an overview of our resource utilization. In addition, we learned how to monitor memory usage, schedule jobs with cron, and gained an understanding of load average.

In [Chapter 7](#), Setting Up Network Services, we'll dive back into networking, where we'll learn how to set up our own DHCP and DNS servers for our network, and more. See you there!

Questions

1. Which command allows you to view running processes on your server?
2. The _____ command shows you a list of running background processes.
3. You can immediately end a process with the _____ command.
4. _____ is a system monitor program that you can optionally install which shows extensive information on current resource utilization.
5. Which utility would you use to start, stop, or restart a process?
6. The _____ command prints information regarding current memory usage.
7. Which command would you use to edit your crontab?

Further reading

- **A Quick and Easy Guide to tmux:** <http://www.hamvoeke.com/blog/a-quick-and-easy-guide-to-tmux/>
- **Tmux Cheat Sheet & Quick Reference:** <https://tmuxcheatsheet.com/>
- **Linux ate my RAM:** <https://www.linuxatemyram.com/>

Setting Up Network Services

In [Chapter 4](#), Connecting to Networks, we discussed various concepts related to networking. We saw how to set the hostname, manage network interfaces, configure connections, use network manager, 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
- Creating a secondary (slave) DNS server
- Setting up an internet gateway
- Keeping your clock in sync with NTP

Planning your IP address scheme

The first step in rolling out any solution is to plan it properly. Planning out your network layout is one of the most important decisions you'll ever make in your organization. Even if as an administrator you're not responsible for the layout and just go along with what your network administrator provides, understanding this layout and being able to deploy your solutions to fit within it is also very important.

Planning an IP address scheme 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're 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, there's always room for growth if your organization is doing well, which is another thing to take into consideration.

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 users on your network, you may think that you're all set. 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 quite quickly 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), 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.

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 (this book is primarily about servers and not a course on network administration), I mentioned 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 cinch 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-bit network, 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
```

Of course, no 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. If you're accustomed to the classful model, this is the same as a class C network. The address 192.168.1.0 refers to the network itself, and that IP is not assignable to clients. In fact, 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.

Next, 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 which 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 (octet) of each IP. This is just personal preference.

Next, we have a block of IP addresses for servers, 192.168.1.11 through 192.168.1.99. 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. As I mentioned, I typically 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 call these 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 be a good idea to separate them, 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 creating a spreadsheet to keep track of your static IP assignments. It doesn't have to be anything fancy, but it will certainly help you later. Ideally, your IP layout and the devices that populate it would be best placed within an internal wiki or knowledge-base solution that you may already be using. But if you don't have anything like that set up yet, a spreadsheet can serve a similar purpose. In my example IP spreadsheet, I include a designation of **(R)** if the IP address is a reservation or **(S)** if the IP address is a manually assigned static address. I also include the MAC address of each device, which will come in handy when we set up our DHCP server in the next section:

Servers

Machine Name	IP	Mac Address 1	Model
D-Link AirPremier DAP-2695	10.10.96.2 (S)	NA	DAP-2695
galaxy	10.10.96.4 (R)	E0:3B:49:6E:6C:8E	Ratel Performance
nagios	10.10.96.3 (R)	52:54:01:88:F8:BC	KVM VM
hermes	10.10.96.1 (S)	00:23:4D:A5:F2:EB	Intel Atom
iris	10.10.96.5 (R)	52:54:01:D5:5D:0C	KVM VM
moogle	10.10.96.102 (R)	52:54:00:D4:3E:87	KVM VM
pi-dev	10.10.96.8 (R)	B8:27:EB:E2:29:03	Raspberry Pi 2
pluto	10.10.96.10 (R)	D0:51:99:37:A9:0D	custom i3 build
upsmon	10.10.96.7 (R)	B8:27: EF :BB:91:2D	Raspberry Pi 2

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.

Serving IP addresses with `isc-dhcp-server`

While most network appliances you purchase nowadays often come with their own DHCP server, rolling your own gives you ultimate flexibility. Some of these built-in DHCP servers are full-featured and come with everything you need, while others may contain just enough features for it to function, but nothing truly exciting. Ubuntu servers make great DHCP servers, and rolling your own server is actually very easy to do.

First, the server that serves DHCP will 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.



If you have yet to set a static IP, [Chapter 4](#), 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
```

Depending on your configuration, the `isc-dhcp-server` service may have started automatically, or your server may have attempted to start it and failed to do so with an error. You can check the status of the daemon with the following command:

```
| systemctl status isc-dhcp-server
```

The output will either show that the service failed or is running. If the service failed to start, that's perfectly fine—we haven't even configured it yet. If it's running, then you need to stop it for now, since it's not a good idea to leave an unconfigured DHCP server running on your network. It might conflict with your existing one.

```
| sudo systemctl stop isc-dhcp-server
```

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 to 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 86400;
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 with 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, the `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 informing the client to use `192.168.1.255` as the broadcast address, which is typically the last address in the subnet and cannot be assigned to a host.

```
| option domain-name "local.lan";
```

Here, we're setting the domain name 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 for multiple networks (which is rare), the `authoritative;` option should be included in your `config` file.

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. We're also providing a default gateway (`option routers`) and DNS server 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. 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.

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:

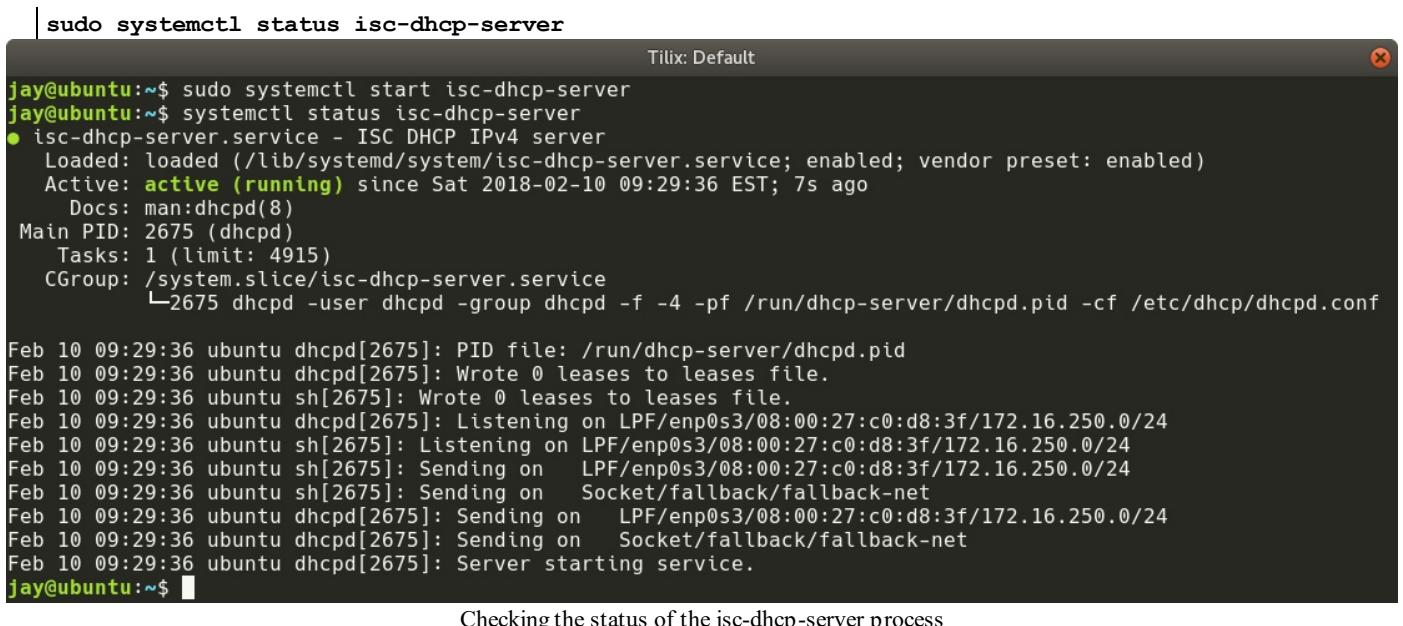
```
| INTERFACESv4="enp0s3"
```

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; the daemon should report that it's `active (running)`, similar to what's shown in the following screenshot (note that I use a different IP address scheme on this network):



The screenshot shows a terminal window titled "Tilix: Default" running on an Ubuntu system. The user has run the command `sudo systemctl status isc-dhcp-server`. The output shows the service is active and running since Saturday, February 10, 2018, at 09:29:36 EST. It lists the main PID (2675) and tasks (1). The service is part of the `/system.slice/isc-dhcp-server.service` unit. The log output shows the DHCP daemon starting and listening on port 67. The log entries from Feb 10 09:29:36 include messages about PID files, lease writes, and socket sending.

```
jay@ubuntu:~$ sudo systemctl start isc-dhcp-server
jay@ubuntu:~$ systemctl status isc-dhcp-server
● 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 Sat 2018-02-10 09:29:36 EST; 7s ago
    Docs: man:dhcpd(8)
 Main PID: 2675 (dhcpd)
   Tasks: 1 (limit: 4915)
  CGroup: /system.slice/isc-dhcp-server.service
          └─2675 dhcpd -user dhcpd -group dhcpd -f -4 -pf /run/dhcp-server/dhcpd.pid -cf /etc/dhcp/dhcpd.conf

Feb 10 09:29:36 ubuntu dhcpd[2675]: PID file: /run/dhcp-server/dhcpd.pid
Feb 10 09:29:36 ubuntu dhcpd[2675]: Wrote 0 leases to leases file.
Feb 10 09:29:36 ubuntu sh[2675]: Wrote 0 leases to leases file.
Feb 10 09:29:36 ubuntu dhcpd[2675]: Listening on LPF/enp0s3/08:00:27:c0:d8:3f/172.16.250.0/24
Feb 10 09:29:36 ubuntu sh[2675]: Listening on LPF/enp0s3/08:00:27:c0:d8:3f/172.16.250.0/24
Feb 10 09:29:36 ubuntu sh[2675]: Sending on   LPF/enp0s3/08:00:27:c0:d8:3f/172.16.250.0/24
Feb 10 09:29:36 ubuntu sh[2675]: Sending on   Socket/fallback/fallback-net
Feb 10 09:29:36 ubuntu dhcpd[2675]: Sending on   LPF/enp0s3/08:00:27:c0:d8:3f/172.16.250.0/24
Feb 10 09:29:36 ubuntu dhcpd[2675]: Sending on   Socket/fallback/fallback-net
Feb 10 09:29:36 ubuntu dhcpd[2675]: Server starting service.
jay@ubuntu:~$
```

Checking the status of the isc-dhcp-server process

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
```

The `-f` flag of the `tail` command is indispensable, and it is 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 (the name of my DHCP server is `hermes`, in case you were wondering):

```
| May  5 22:07:36 hermes dhcpcd: DHCPDISCOVER from 52:54:00:88:f8:bc via enp0s3
| May  5 22:07:36 hermes dhcpcd: 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/dhcpcd.leases` file, and a typical lease entry in that file would look similar to the following:

```
lease 192.168.1.138 {
    starts 0 2016/05/06 16:37:30;
    ends 0 2016/05/06 16:42:30;
    cltt 0 2016/05/06 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 into 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/dhcpcd.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 without having to plug a monitor in to it. Similarly, you can view the temporary IP address of a new network appliance 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 (the `/etc/network/interfaces` file), as well as within your `/etc/dhcp/dhcpcd.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.

Setting up DNS with bind

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/> with 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 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 address 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.

First, we'll need to install the **Berkeley Internet Name Daemon (BIND)** package on our server:

```
| sudo apt install bind9
```

At this point, we 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 the 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 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. It should report that it's `active (running)`:

```
| systemctl status bind9
```

```
Tilix: Default  
● bind9.service - BIND Domain Name Server  
  Loaded: loaded (/lib/systemd/system/bind9.service; enabled; vendor preset: enabled)  
  Active: active (running) since Sat 2018-02-10 09:26:56 EST; 7min ago  
    Docs: man:named(8)  
   Process: 2644 ExecStop=/usr/sbin/rndc stop (code=exited, status=0/SUCCESS)  
 Main PID: 2647 (named)  
    Tasks: 4 (limit: 4915)  
   CGroup: /system.slice/bind9.service  
         └─2647 /usr/sbin/named -f -u bind  
  
Feb 10 09:26:56 ubuntu named[2647]: managed-keys-zone: journal file is out of date: removing journal  
Feb 10 09:26:56 ubuntu named[2647]: managed-keys-zone: loaded serial 2  
Feb 10 09:26:56 ubuntu named[2647]: zone 0.in-addr.arpa/IN: loaded serial 1  
Feb 10 09:26:56 ubuntu named[2647]: zone 127.in-addr.arpa/IN: loaded serial 1  
Feb 10 09:26:56 ubuntu named[2647]: zone localhost/IN: loaded serial 2  
Feb 10 09:26:56 ubuntu named[2647]: zone 255.in-addr.arpa/IN: loaded serial 1  
Feb 10 09:26:56 ubuntu named[2647]: all zones loaded  
Feb 10 09:26:56 ubuntu named[2647]: running  
Feb 10 09:26:56 ubuntu named[2647]: managed-keys-zone: Key 19036 for zone . acceptance timer complete  
Feb 10 09:26:56 ubuntu named[2647]: managed-keys-zone: Key 20326 for zone . acceptance timer complete  
lines 1-20/20 (END)
```

Checking the status of the bind9 service

As long as you've entered everything correctly, you should now have a working DNS

server. Of course, it isn't resolving anything, 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 lines 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 your new server. 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 that is utilizing your new DNS server:

```
| dig www.packtpub.com
```

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.

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 what is known as 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. (
    201808161 ; 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 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:

```
| @ IN SOA local.lan. hostmaster.local.lan. (
```

With the **Start of Authority (SOA)** line, we're establishing that our DNS server is authoritative over the `local.lan` domain. 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, that's not an issue we'll need to worry about:

```
| 201808161 ; serial
```

Of all the lines of configuration within a zone file, the `serial` is by far the one that will frustrate us the most. 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 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 the serial many times and became frustrated when the DNS server refused to resolve new hosts that were recently added. 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. The format doesn't really matter; I used `201808161`, 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 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:

```
8H ; refresh  
4H ; retry  
4W ; expire  
1D ) ; minimum
```

These values control how often slave DNS servers will be instructed to check in for updates. With the `refresh` value, we're instructing any slave 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 slave 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:

```
IN A 192.168.1.1  
@ IN NS hermes.local.lan.
```

Here, we identify the name server itself. 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. In my 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:

```
| 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 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  
Tilix: Default  
● bind9.service - BIND Domain Name Server  
  Loaded: loaded (/lib/systemd/system/bind9.service; enabled; vendor preset: enabled)  
  Active: active (running) since Sat 2018-02-10 09:48:29 EST; 4min 28s ago  
    Docs: man:named(8)  
   Process: 2857 ExecStop=/usr/sbin/rndc stop (code=exited, status=0/SUCCESS)  
 Main PID: 2860 (named)  
    Tasks: 4 (limit: 4915)  
   CGroup: /system.slice/bind9.service  
         └─2860 /usr/sbin/named -f -u bind  
  
Feb 10 09:48:29 ubuntu named[2860]: zone 0.in-addr.arpa/IN: loaded serial 1  
Feb 10 09:48:29 ubuntu named[2860]: zone 255.in-addr.arpa/IN: loaded serial 1  
Feb 10 09:48:29 ubuntu named[2860]: zone 127.in-addr.arpa/IN: loaded serial 1  
Feb 10 09:48:29 ubuntu named[2860]: zone localhost/IN: loaded serial 2  
Feb 10 09:48:29 ubuntu named[2860]: zone local.lan/IN: loaded serial 201808162  
Feb 10 09:48:29 ubuntu named[2860]: all zones loaded  
Feb 10 09:48:29 ubuntu named[2860]: running  
Feb 10 09:48:29 ubuntu named[2860]: zone local.lan/IN: sending notifies (serial 201808162)  
Feb 10 09:48:29 ubuntu named[2860]: managed-keys-zone: Key 19036 for zone . acceptance time  
Feb 10 09:48:29 ubuntu named[2860]: managed-keys-zone: Key 20326 for zone . acceptance time  
lines 1-20/20 (END)
```

Checking the status of the bind9 service after adding a new zone

You should see that the service state is `active (running)`, and in addition a line telling you that the serial number for your zone file was loaded. In my case, serial number `201808162` was loaded for zone `local.lan`. If you don't see the service is 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 and external resource. For example, you could try `dig` against a URL as well as a local server:

```
| 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 2018  
|;; MSG SIZE rcvd: 83
```

What you're looking for here is that both local resources and external websites should be resolvable now. You'll probably notice that the DNS server used in the output will most likely show up as a localhost address, such as it did in my output and not 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.

Creating a secondary (slave) DNS server

Depending on just one server to provide a resource to your network is almost never a good idea. If our DNS server has a problem and fails, our network users will be unable to resolve any names, internal or external. To rectify this, we can actually set up a slave DNS server that will cache zone records from the master and allow name resolution to work in case the primary fails. This is not required, but redundancy is always a good thing.

To set up a secondary DNS server, we first need to configure our primary server to allow it to transfer zone records to a slave server. To do so, we'll need to edit the `/etc/bind/named.conf.options` file, which currently looks similar to the following:

```
options {
    directory "/var/cache/bind";
    forwarders {
        8.8.8.8;
        8.8.4.4;
    };
    dnssec-validation auto;

    auth-nxdomain no;
    listen-on-v6 { any; };
};
```

I've omitted some redundant lines from the file (such as comments). When we edited this file the last time, we uncommented the forwarders section and added two external DNS servers there. In order to allow the transfer of zone records to another server, we'll need to add a new line to this file. Here's the contents of this file again, with a new line added to it:

```
options {
    directory "/var/cache/bind";
    allow-transfer { localhost; 192.168.1.2; };
    forwarders {
        8.8.8.8;
        8.8.4.4;
    };
    dnssec-validation auto;

    auth-nxdomain no;
    listen-on-v6 { any; };
};
```

Basically, we added one line to the file, which allows zone transfer to an IP address of

`192.168.1.2`, which you'd want to change to be the IP address for your secondary DNS server. Next, restart the `bind9` service on your primary server, and it should be ready to allow transfer to another machine.

On the slave DNS server, the first thing you'll need to do is to install the `bind9` packages as we did with the first server. Then, we'll configure the slave server quite similar to the master server, but with some notable differences. On the slave server, you'll omit the `allow-transfer` line that we added to the `/etc/bind/named.conf.options` file, since the slave server is only going to receive records, not transfer them. Like before, you'll uncomment the `forwarders` section in the `/etc/bind/named.conf.options` file. In the `/etc/bind/named.conf.local` file, you'll add an entry for our zone just as we did with the first server, but we'll configure this file differently than we did the last time:

```
zone "local.lan" IN {
    type slave;
    masters { 192.168.1.1; };
    file "/var/lib/bind/net.local.lan";
};
```

Inside the `/etc/bind/named.conf.local` file on the slave server, we first identify it as a slave (`type slave;`) and then we set the master server to `192.168.1.1` (or whatever the IP address of your primary DNS server is). Then, we save our zone file to a different location than we did on the master; in this case, it is `/var/lib/bind/net.local.lan`. This is due to how the permissions differ on the slave and the master server, and `/var/lib/bind` is a good place to store this file. After the new configuration is in place, restart `bind9` on both the slave and the master servers. At this point though, we should test both servers out one last time to ensure both are working properly.

In this chapter, I've mentioned the `dig` command a few times. It's a great way to interrogate DNS servers to ensure they're providing the correct records. One thing I haven't mentioned so far is that you can specify a DNS server for the `dig` command to check, rather than having it use whatever DNS server the system was assigned. In my examples, I have two DNS servers set up, one at `192.168.1.1` and the other at `192.168.1.2`. Specifying a DNS server to interrogate with the `dig` command is easy enough:

```
dig @192.168.1.1 fileserv
dig @192.168.1.2 fileserv
```

You should see results from both of your DNS servers. When you check the status of the `bind9` daemon on your slave server (`systemctl status bind9`), you should see entries that indicate a successful transfer. The output will look similar to this:

```
May 06 13:19:47 ubuntu-server named[2615]: transfer of 'local.lan/IN' from 10.10.99.184#53: connection
May 06 13:19:47 ubuntu-server named[2615]: zone local.lan/IN: transferred serial 201602093
May 06 13:19:47 ubuntu-server named[2615]: transfer of 'local.lan/IN' from 10.10.99.184#53: Transfer complete
May 06 13:19:47 ubuntu-server named[2615]: transfer of 'local.lan/IN' from 10.10.99.184#53: Transfer complete
May 06 13:19:47 ubuntu-server named[2615]: zone local.lan/IN: sending notifies (serial 201602093)
```

From this point forward, you should have a redundant `bind` slave to work with, but the server is useless until you inform your clients to use it. To do that, we need to revisit our DHCP server configuration yet again. Specifically, the `/etc/dhcp/dhcpd.conf` file. Here are the sample contents of the file again, so you don't need to flip back to the earlier section:

```
default-lease-time 86400;
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;
}
```

On the last line, we inform clients to set their DNS server to `192.168.1.1`. However, since we now have a slave server, we should change this file to include it so that it's passed along to clients:

```
default-lease-time 86400;
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, 192.168.1.2;
}
```

Basically, all we did was add the secondary server to the `option domain-name-servers` line. We separated the first with a comma, then we ended the line with a semicolon as we did before. At this point (assuming you've tested your secondary DNS server and trust it), you can restart the `isc-dhcp-server` daemon, and your nodes will have the primary and secondary DNS servers assigned to them the next time they check in for an IP address or to renew their existing lease.

That about does it for our journey into setting up `bind`. From now on, you have a DHCP server and a DNS server or two in your network, providing addressing and name resolution. We've pretty much set up two of the most common services that run on

commercial routers, but we haven't actually set up a router just yet. In the next section though, we'll change that.

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. In most commercial routers, we'll have DNS and DHCP built in, as well as routing. Quite often, these services will all run on the same box. Depending on how you set up your DNS and DHCP servers in the previous sections, you may have even set up your primary DNS and DHCP servers on the same machine, which is quite common. However, your internet connection will likely be terminated on a separate box, possibly a commercial routing device or internet gateway from your internet service provider.

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.

Linux servers handle this job very well, and if you want to consolidate your DHCP, DNS, and routing into a single server, that's a very easy (and common) thing to do. Specifically, you'll need a server with at least two Ethernet ports, as well as a network switch that will allow you to connect multiple devices. If you need to connect devices with wireless network cards, you'll need an access point as well. 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.

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 in to common routing equipment are extremely poor and most of them are easy to hack when someone wants in bad 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, as well as installing an authentication monitor such as `fail2ban`. We'll get into those topics in [Chapter 15](#), 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 on 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 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!

Keeping your clock in sync with NTP

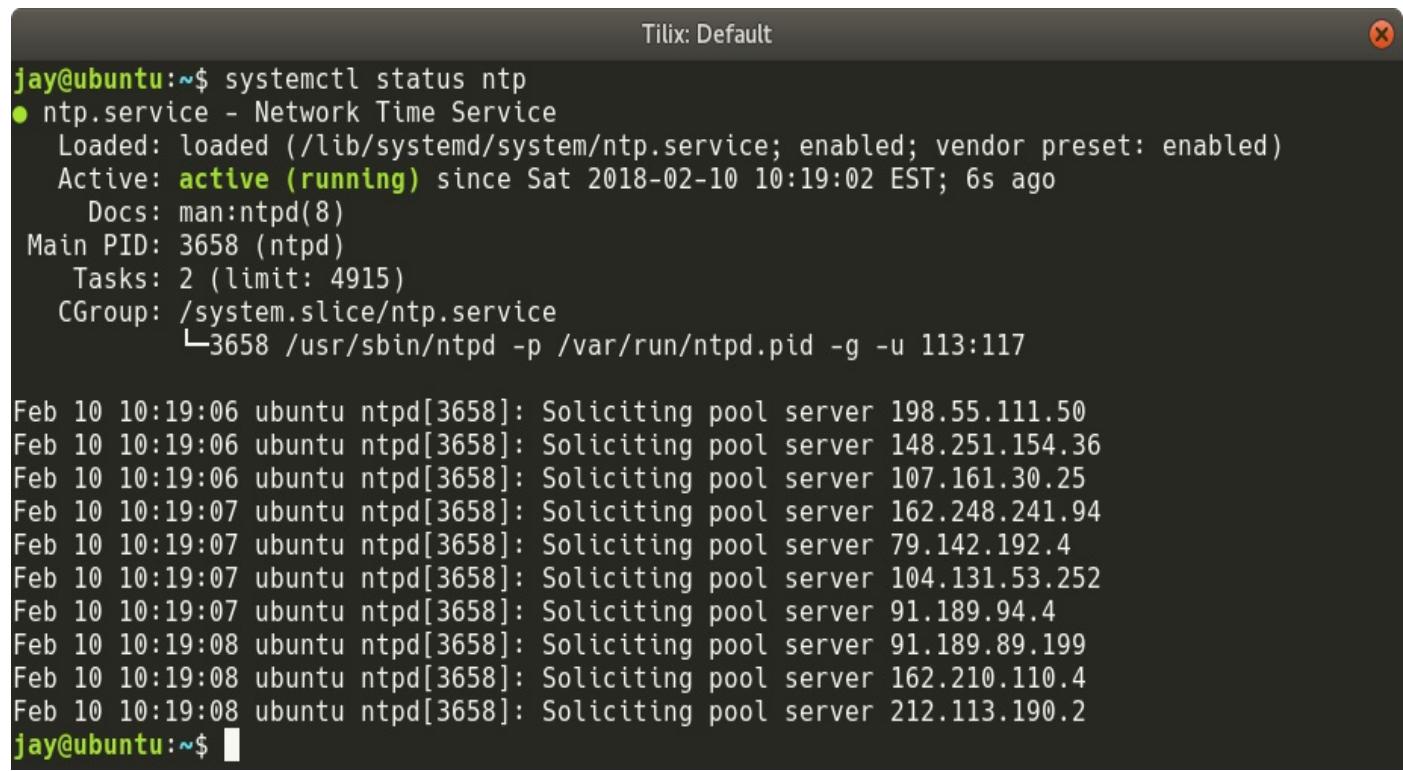
It's incredibly important for Linux servers to keep their time synchronized, as strange things can happen when a server's clock is wrong. One issue I've run into that's especially problematic is file synchronization utilities, which will exhibit strange behavior when there are time issues. However, Ubuntu servers feature the NTP client and server within the default repositories to help keep your time in sync. If it's not already installed, all you should need to do is install the `ntp` package:

```
| sudo apt install ntp
```

Once installed, the `ntp` daemon will immediately start and will keep your time up to date. To verify, check the status of the `ntp` daemon with the following command:

```
| systemctl status ntp
```

The output should show that `ntp` is running:



A screenshot of a terminal window titled "Tilix: Default". The terminal shows the command `systemctl status ntp` being run by user "jay@ubuntu". The output indicates that the `ntp.service` is active (running) since February 10, 2018, at 10:19:02 EST. It lists various details such as the main PID (3658), tasks (2), and cgroup (/system.slice/ntp.service). Below this, a log of NTP pool solicitations is displayed, listing several IP addresses from which the service is soliciting time.

```
jay@ubuntu:~$ systemctl status ntp
● ntp.service - Network Time Service
  Loaded: loaded (/lib/systemd/system/ntp.service; enabled; vendor preset: enabled)
  Active: active (running) since Sat 2018-02-10 10:19:02 EST; 6s ago
    Docs: man:ntpd(8)
 Main PID: 3658 (ntpd)
   Tasks: 2 (limit: 4915)
  CGroup: /system.slice/ntp.service
          └─3658 /usr/sbin/ntpd -p /var/run/ntpd.pid -g -u 113:117

Feb 10 10:19:06 ubuntu ntpd[3658]: Soliciting pool server 198.55.111.50
Feb 10 10:19:06 ubuntu ntpd[3658]: Soliciting pool server 148.251.154.36
Feb 10 10:19:06 ubuntu ntpd[3658]: Soliciting pool server 107.161.30.25
Feb 10 10:19:07 ubuntu ntpd[3658]: Soliciting pool server 162.248.241.94
Feb 10 10:19:07 ubuntu ntpd[3658]: Soliciting pool server 79.142.192.4
Feb 10 10:19:07 ubuntu ntpd[3658]: Soliciting pool server 104.131.53.252
Feb 10 10:19:07 ubuntu ntpd[3658]: Soliciting pool server 91.189.94.4
Feb 10 10:19:08 ubuntu ntpd[3658]: Soliciting pool server 91.189.89.199
Feb 10 10:19:08 ubuntu ntpd[3658]: Soliciting pool server 162.210.110.4
Feb 10 10:19:08 ubuntu ntpd[3658]: Soliciting pool server 212.113.190.2
jay@ubuntu:~$
```

Checking the status of the NTP service

If all you wanted was a working NTP client, then you're actually all set. The default configuration is fine for most. But further explanation will help you understand how this

client is configured. You'll find the configuration file 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. However, if you have a great number of clients in your organization, it may make sense to set up your own NTP server so that only one server is talking to the outside world instead of every one of your nodes. Essentially, this is how you can become a friendly neighbor online. If you have hundreds of workstations in your organization that are configured to check Ubuntu's NTP servers, that's quite a bit of traffic those servers will need to endure. To be fair, Ubuntu's servers won't have any trouble handling this traffic, though it's a nice gesture to configure one server in your network to synchronize with Ubuntu's time servers, then configure your local nodes to synchronize with just your local server. If everyone did this, then Ubuntu's time servers would see far less traffic.

Setting up a local NTP server is actually quite straightforward. All you need to do is designate a server within your network for this purpose. You could even use the internet gateway you set up in the previous section if you wanted to. Once you have that server set up to synchronize, you should be able to configure your local nodes to talk to that server and synchronize their clocks with 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:

Tilix: Default

```

jay@ubuntu:~$ 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
#ntp01.plutex.de 131.188.3.223  2 u   60  64  377  164.471  -22.743  21.259
-69.195.159.158 128.138.140.44  2 u   58  64  377   50.374   6.445  4.136
-tick.no-such-ag 162.213.2.253  2 u   56  64  377   80.940   1.497  2.939
-clocka.ntpjs.or 35.73.197.144  2 u   53  64  377   34.591  -1.738  3.140
-eva.aplu.fr     145.238.203.14  2 u   51  64  377   119.085   2.020  27.310
+195.50.171.101 145.253.2.212  2 u   57  64  377   135.565  -5.086  14.310
-us-ga.ntp.dark- 127.67.113.92  2 u   55  64  377   38.150   0.073  3.097
#services.quadra 74.117.214.3   2 u   50  64  377   82.351  -7.825  2.472
-s1.vlns.de      94.199.173.113  2 u   54  64  377   117.312  -2.801  21.898
#82.193.117.90 ( 31.28.161.68  2 u   55  64  377   182.481  -26.820  19.104
+104.131.53.252 209.51.161.238  2 u   51  64  377   28.824  -4.458  2.763
+ns1.backplanedn 71.176.109.169  2 u   56  64  377   50.853  -6.505  2.093
#pugot.canonical 140.203.204.77  2 u   64  64  377   195.515  -46.154 38.454
#ns1.infomir.com 243.50.127.182  2 u   56  64  377   183.797  -2.271  24.432
*resolver1.skyfi 216.218.192.202  2 u   49  64  377   72.254  -5.821  3.361
#golem.canonical 17.253.34.253   2 u    6  64  377   177.403  -21.229 16.522
#a212-113-190-2. 10.176.63.114  3 u   54  64  377   229.731  -44.193 33.955

```

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, specifically whether the server is using **unicast**, **broadcast**, **multicast**, or **multicast**.

Continuing, the `when` column refers to how long ago it was 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. This number is in octal, so eight successes in octal will be represented by ³⁷⁷. When you first start the `ntp` daemon on your server, it may take some time for this number to reach ³⁷⁷.

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. Once you restart the `ntp` service on your other nodes, they should start synchronizing with the master server.

Before we close out this chapter, there is one additional change you should consider implementing to the `/etc/ntp.conf` file, however. 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. 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, as well as plan a pool of addresses for DHCP. We also set up a DHCP and DNS server, with an additional section on creating a DNS slave for added redundancy. We closed off this chapter with discussions on setting up an internet gateway, as well as configuring NTP.

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!

Questions

1. What is subnetting, and how does it benefit your network?
2. The first step in setting up a DHCP server is installing the _____ package.
3. In what file are DHCP leases stored?
4. What benefit does a DNS server provide your network?
5. What is the benefit of a secondary DNS server?
6. Which command can you use to interrogate a DNS server?
7. _____ helps you keep your system clock in sync with an external time server.

Further reading

- **8 Steps to Understanding IP Subnetting:** <https://www.techopedia.com/6/28587/internet/8-steps-to-understanding-ip-subnetting>

Sharing and Transferring Files

Within an enterprise network, having one or more servers available to store files and make data accessible over the network is a great asset. 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 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

File server considerations

There are two common technologies you can utilize to share files with your users, **Samba** and **NFS**. In fact, there's nothing stopping you from hosting both Samba and NFS shares on a single server. 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 of standard UNIX permissions is provided, so you'll only need to be able to configure your permissions once. If permissions and confidentiality are important to you, you may want to look closer at NFS.

That's not 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. You'll need to enable a

specific plugin for your version of Windows in order to access NFS shares. The problem is that historically Microsoft has limited access to these tools to only the more expensive Windows editions, such as Ultimate and Enterprise when it comes to Windows 7, and Enterprise when it comes to Windows 8. Thankfully though, Microsoft came to their senses a bit and have allowed the Linux Subsystem to be installed on any version of Windows 10, so this licensing restriction only comes into play on older versions. Depending on how many legacy Windows machines exist in your environment that need to access NFS shares, this could be a significant licensing cost to your organization. This is yet another reason why Samba is a great choice when you're dealing with the Windows platform; you can circumvent the cost of upgraded licenses, since even home editions of legacy versions of Windows can access Samba shares.

In regards to an all-Linux environment or in a situation where you only have Linux machines that need to access your shares, NFS is a great choice because its integration is much tighter 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 on to read 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 `/shared` directory at the `root` filesystem and create sub-directories for my network shares within it. For example, I can create `/shared/documents`, `/shared/public`, and so on for Samba shares. With regards 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.

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 hosts 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 hosts wins
```

The `name resolve order` setting configures how Samba resolves hostnames. In this case, we're using the broadcast name first, followed by any mappings that might exist in our `/etc/hosts` file, followed by `wins`. Since `wins` has been pretty much abandoned (and replaced by DNS), we include it here solely for compatibility.

```
| 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 into its 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 the 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 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 intent 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 `/shared/documents` and `/shared/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.

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.

With `testparm` out of the way, feel free to start Samba:

```
| sudo systemctl start smbd
```

That really should be all there is to it; you should now have a `Documents` and `Public` share 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, 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. Otherwise, you can also access a Samba share by adding an entry for it in the `/etc/fstab` file, such as the following:

```
| //myserver/shared/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), 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 the 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
```

Setting up NFS shares

A **Network File System (NFS)** is 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 there may be an additional licensing penalty if you need to upgrade to a different release. NFS is preferred in a Linux or UNIX environment though, since it fully supports Linux- and UNIX-style permissions. As you can see from our earlier dive into Samba, 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. Samba can certainly support per-user access restrictions and benefit 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.

Earlier, we set up a parent directory on 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 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 directory 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 to 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. We can set here 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 export
```

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 Users, 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:

```
| # sudo 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 7](#), 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 -l 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; 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:

```
| 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. 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 for the user to have to memorize the fact that the server is sharing a documents 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, anytime you add a new export, they 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 on 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 with the man page on `export`. 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 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, 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 to where you send it. I'll walk you through some `rsync` examples that will progressively increase in complexity. The first few examples will show you how to backup 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 probably make a habit of including this if you use no other option). You should now see a copy of the `myuser` home 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 instead 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 in the industry. 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 the `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 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 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. Here, we have two 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 set it to a variable (`CURDATE`). We'll also `export` the new variable so that it is available everywhere. In the `rsync` portion of the command, we use the variable for the `--backup-dir` option. This will copy replaced files to a `backup` directory named after the `date` the command was run. Basically, if today's date was `08-17-2018`, the resulting command would be the same as if we run the following:

```
| sudo rsync -avb --delete --backup-dir=/backup/incremental/08-17-2018 /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 OpenSSH. 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 more 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. To utilize SCP, we'll use the `scp` command. Since you most likely already have OpenSSH 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. 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 as well:

```
| 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 the file is where you expect it to be 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 just 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 regards 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 15](#), Securing Your Server.

Like most commands in the Linux world, the `scp` command supports the 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, 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 as well as 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 use 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 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 he or she 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 lazy, 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 9](#), 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!

Questions

1. What are two primary technologies used to set up file shares on a network?
2. _____ is a technology that utilizes OpenSSH and allows users to mount directories over an SSH connection.
3. _____ and _____ are two command-line utilities that can be used to send files over the network.
4. When is Samba a good choice for setting up a file share?
5. When is NFS a good choice for setting up a file share?
6. Which file can be used with NFS to better map usernames?
7. The _____ command can be used to unmount an SSHFS attachment.

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/SettingUpNFSTo>

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 master/slave 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 slave 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 you are probably familiar with MySQL. MySQL is a tried and true solution which 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 were paying attention to the news in the open source community over the last several years, 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 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. The Linux community, in general, doesn't seem to trust Oracle as a company, and in turn doubts its stewardship of MySQL. I'm not going to get into any speculation about Oracle, the future of MySQL, or any conspiracies 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.

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, and new technologies tend to scare management. 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, most of my instructions should work for MySQL as well.

With regards 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 new found 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 slave 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 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 work day 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 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 3](#), 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 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 only 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 `mysql` for `mariadb` whenever I mention it if you are using MySQL instead. 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 following command still references `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 1045 (28000): Access denied for user 'root'@'localhost' (using password: YES)

You have two options for dealing with this. First, you can simply use `sudo mariadb` to access the MariaDB shell and not use this 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:
mysql -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` 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 slave 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 slave), 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 MariadDB 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 SIGHUP trap that is executed if the `mariadb` process receives the SIGHUP signal, which will allow 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 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 the `/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 comes time to set up master and slave replication.

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 contents 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, even though the order the configuration files are checked is set to check the `mariadb.cnf` file first, followed by the `/etc/mysql/conf.d` and `/etc/mysql/mariadb.conf.d` directories, the `mariadb.cnf` file simply tells the `mariadb` service to immediately check those 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 the 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 setting up our master/slave 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 implement 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.

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 create this administrative user, we'll need to enter the MariaDB shell, which again is simply a matter of executing the `mariadb` command with `sudo`. Alternatively, we can use the following command as a normal user to switch to `root`, without using `sudo`:

```
mariadb -u root -p
```

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 privilege 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 he or she 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). 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 DB 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 afterwards), 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 mysampled;
```

That was easy. We should now have a database on our server named `mysampled`. To list all databases on our server (and confirm our database was created properly), we can execute the following command:

```
| SHOW DATABASES;
```

jay@ubuntu:~

```
MariaDB [(none)]> SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| mysampled      |
| mysql          |
| performance_schema |
+-----+
4 rows in set (0.01 sec)

MariaDB [(none)]>
```

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;
```

jay@ubuntu:~

```
MariaDB [(none)]> SELECT HOST, USER, PASSWORD FROM mysql.user;
+-----+-----+-----+
| HOST    | USER   | PASSWORD           |
+-----+-----+-----+
| localhost | root   | *33A30BF3798B7E5E5EC469931C44C650889BD231 |
| 192.168.1.% | admin  | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
| 192.168.1.% | jay    | *2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19 |
+-----+-----+-----+
3 rows in set (0.00 sec)

MariaDB [(none)]>
```

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 as little permission as required to allow it to function properly. We've already created the `mysampled` 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 mysampled.* 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 `mysampled` database. This is equivalent to read-only access. If our user needed full access, we could use the following instead:

```
| GRANT ALL ON mysampled.* 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 `mysampled` 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 access 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've 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 `mysampled` 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 `mysampled` database. Now, we can have some fun and modify the contents. Here are some examples you can follow:

```
| USE mysampled;
```

The `USE` command allows us to select a database we want to work with. The MariaDB prompt will change from `MariaDB [(none)]>` to `MariaDB [mysampled]>`. 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 `mysampled` 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;
```

```
jay@ubuntu:~  
MariaDB [mysampled] > SELECT * FROM Employees;  
+-----+-----+-----+  
| Name      | Age   | Occupation |  
+-----+-----+-----+  
| Joe Smith |    26 | Ninja      |  
| Fox Mulder |    55 | FBI Agent   |  
| Bruce Wayne |    45 | The Batman  |  
+-----+-----+-----+  
3 rows in set (0.00 sec)  
  
MariaDB [mysampled]>
```

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 mysampled;
```

There is, of course, much more to MariaDB and its MySQL syntax than the samples I 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 on 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 mysampledbs > mysampledbs.sql
```

With this example, we're using `mysqldump` to create a copy of the `mysampledbs` database and storing it in a file named `mysampledbs.sql`. Since MariaDB requires us to log in, we authenticate to MariaDB using the `-u` option with the username `admin` and the `-p` option that 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 `mysampledbs` database will be dumped into the `mysampledbs.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 < mysampledbs.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 slave database server.

Setting up a slave database server

Earlier in this chapter, we discussed installing MariaDB on a server. To set up a slave, all you really need to begin the process is set up another database server. If you've already set up two database servers, you're ready to begin. If not, feel free to spin up another VM and follow the process from earlier in this chapter that covered installing 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 master and the other the slave, so make a note of the IP addresses for each.

To begin, we'll first start working on the master. We'll need to edit the `/etc/mysql/conf.d/mysql.cnf` file. 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=mysampled
server-id=1
```

With this configuration, we're first enabling bin logging, which is required for a master/slave server to function properly. These binary logs record changes made to a database, which will then be transferred to a slave.

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 slave). Change this line to the following:

```
| bind-address = 0.0.0.0
```

Next, we'll need to access the MariaDB shell on the master and execute the following commands:

```
| GRANT REPLICATION SLAVE ON *.* to 'replicate'@'192.168.1.204' identified by 'slavepassword';
```

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 slave, 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 `slavepassword`, 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 slave 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 master server. The process of setting up a slave is much easier if we don't have to worry about the master database changing while we set up the slave. 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 master, you can disregard that step.

Next, we should utilize `mysqldump` to make both the master and the slave 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 master server's database and then import that dump into the slave. The easiest way to transfer the dump file is to use `rsync` or `scp`. Then, on the slave, use `mariadb` to import the file.

The command to back up the database on the master becomes the following:

```
| mysqldump -u admin -p --databases mysampledbs > mysampledbs.sql
```

After transferring the `mysampledbs.sql` file to the slave, you can import the backup into the slave server:

```
| mariadb -u root -p < mysampledbs.sql
```

Back on the slave, we'll need to edit the `/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 multiple slaves. If you do, each will need a unique server-id.

Make sure you restart the mariadb unit on the slave before continuing:

```
| sudo systemctl restart mariadb
```

From the root MariaDB shell on your slave, 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='slavepass'
```

Now that we're finished configuring the synchronization, we can unlock the master's tables. On the master server, execute the following command within the MariaDB shell:

```
| UNLOCK TABLES;
```

Now, we can check the status of the slave to see whether or not it is running. Within the slave'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 slave isn't running (Slave_IO_State is blank), execute the following command:

```
| START SLAVE;
```

Next, check the status of the slave again to verify:

```
| SHOW SLAVE STATUS\G;
```

From this point forward, any data you add to your database on the master should be replicated to the slave. To test, add a new record to the Employees table on the mysampled database on the master server:

```
| USE mysampled;
| INSERT INTO Employees VALUES ('Optimus Prime', '100', 'Transformer');
```

On the slave, check the same database and table for the new value to appear. It may take

a second or two:

```
| USE mysampled;
| 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 database master is listening for connections on `0.0.0.0` port `3306`. To test, run this variation of the `netstat` command to see which port the `mariadb` process is listening on (it's listed as `mysqld`):

```
| sudo netstat -tulpn |grep mysql
```

The output should be similar to the following:

tcp	0	0 0.0.0.0:3306	0.0.0.0:*	LISTEN	946/mysqld
-----	---	----------------	-----------	--------	------------

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 slave when you run `SHOW SLAVE STATUS\G`; with regards to authentication, make sure you've run `FLUSH PRIVILEGES`; on the master. 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 master to grant replication permissions:

```
| GRANT REPLICATION SLAVE ON *.* to 'replicate'@'192.168.1.204' identified by 'slavepassword';
| FLUSH PRIVILEGES
```

Here's the command that we ran on the slave:

```
| CHANGE MASTER TO MASTER_HOST="192.168.1.184", MASTER_USER='replicate', MASTER_PASSWORD='slavepass';
```

Finally, make sure that your master database and the slave database both contain the same databases and tables. The master won't be able to update a database on the slave 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 slave once, since after you get the replication going, any changes made to the database on the master should follow over to the slave. If you have any difficulty with the `mysqldump` command, you can manually create the `mysampledb` and the `Employees` table on the slave, 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 slave 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 master and slave servers at your disposal. To get additional practice, try adding additional databases, tables, 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 slave, so you can use the commands we've used earlier in this chapter to create users on the slave 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 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 master and slave 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 10, Serving Web Content](#), where we'll journey into the world of web hosting.

Questions

1. MariaDB is a fork of which popular database engine?
2. When setting up a database server, which command can be used to provide basic security for MariaDB?
3. What command would you run to create a user named `samus` with a password of `sr388` and allow access from any IP address on the `10.100.1.0` network?
4. Assuming you have a database named `MyAppDB`, what command would you use to give user `william` full access to this database with the password `mysecretpassword` and allow connection from any IP address in the `192.168.1.0` network?
5. What utility can you use to copy an entire database to a file?
6. Which command would you use to fetch a list of all user MariaDB user accounts?
7. Which command would you use to remove a database named `MyAppDB`?

Serving Web Content

The flexible nature of Ubuntu Server makes it an amazing platform 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 it with SSL. 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 collaborate on and share 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 SSL
- Installing and configuring NGINX
- Setting up failover with keepalived
- Setting up and configuring Nextcloud

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, which is 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:



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 as far as 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 a **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 was 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 would consist of an individual configuration file, which differentiate themselves 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 a separate configuration file 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 the Ubuntu Server, typically in a subdirectory 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 of `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 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 replaced `reload` with `restart` on 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 the `/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 in to the server from port `80`. The `ServerAdmin` clause specifies the email address that is displayed in any error messages shown in case 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.

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. Here's an example for a hypothetical website, `acmeconsulting.com`:

```
<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>
```

I've emphasized some important differences in this file. 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 the `DocumentRoot` to `/var/www/acmeconsulting`. Each virtual host should have their own individual `DocumentRoot` to keep each site separate from one another. 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 their own directory as a subdirectory 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 their 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 file names, so this virtual host is logging errors in the `/var/log/apache2/acmeconsulting.com-error.log`, 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 file. I called mine `000-virtual-hosts.conf` and stored it in the `/etc/apache2/sites-available` 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 Document Root of `/var/www/acmeconsulting`. The second 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.

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. The easiest way to test virtual hosts is with a virtual machine. Software such as VirtualBox allows you to set up virtual machines with multiple network interfaces. If you set each one to be a bridged connection, they should get an IP address from your local DHCP server, and you can then create virtual hosts to serve different websites depending on which interface the request comes in on.

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 mods, `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-python` (which adds Python support) and `libapache2-mod-php7.2` (which adds PHP 7.2 support), among many others. Installing an Apache module is done the same way as any other package, with the `apt install` or `aptitude` commands. In the case of PHP support, we can install the required package with the following command:

```
| sudo apt install libapache2-mod-php7.2
```

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. We can use the `a2enmod` and `a2dismod` commands for enabling or disabling a module. If you wish to simply view a list of modules that Apache has available, the following command examples will help.

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-in to 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:~$ a2enmod
Your choices are: access_compat actions alias allowmethods asis auth_basic auth_digest a
uth_form authn_anon authn_core authn_dbd authn_dbm authn_file authn_socache authnz_fcgi
authnz_ldap authz_core authz_dbd authz_dbm authz_groupfile authz_host authz_owner authz_
user autoindex buffer cache cache_disk cache_socache cern_meta cgi cgid charset_lite dat
a dav dav_fs dav_lock dbd deflate dialup dir dump_io echo env expires ext_filter file_ca
che filter headers heartbeat heartmonitor http2 ident imagemap include info lbmethod_byb
usyness lbmethod_byrequests lbmethod_bytraffic lbmethod_heartbeat ldap log_debug log_for
ensic lua macro mime mime_magic mpm_event mpm_prefork mpm_worker negotiation php7.2 prox
y proxy_ajp proxy_balancer proxy_connect proxy_express proxy_fcgi proxy_fdpass proxy_ftp
proxy_hcheck proxy_html proxy_http proxy_http2 proxy_scgi proxy_wstunnel ratelimit refl
ector remoteip reqtimeout request rewrite sed session session_cookie session_crypto sess
ion_dbd setenvif slotmem_plain slotmem_shm socache_dbm socache_memcache socache_shmcb sp
eling ssl status substitute suexec unique_id userdir usertrack vhost_alias xml2enc
Which module(s) do you want to enable (wildcards ok)?
```

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 php7.2
```

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-php7.2` package I used as an example, the module should've been enabled for you once the package was installed:

```
| sudo a2enmod php7.2
| Module php7.2 already enabled
```

If the module wasn't already enabled, we would see the following output:

```
| Enabling module php7.2.
| To activate the new configuration, you need to run:
|   service apache2 restart
```

As instructed, we'll need to restart Apache in order for the enabling of a module to take effect. The same also holds true when we disable a module as well. Disabling modules works pretty much the same way, you'll use the `a2dismod` command along with the name of the module you'd like to disable:

```
| sudo a2dismod php7.2
| Module php7.2 disabled.
| To activate the new configuration, you need to run:
|   service apache2 restart
```

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 SSL

Nowadays, it's a great idea to ensure your organization's website is encrypted and available over HTTPS. Encrypting your web traffic is not that hard to do and will help protect your organization against common exploits. Utilizing SSL 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 SSL with our Apache installation. We'll work through enabling SSL, 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 netstat -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:

```
| tcp6      0      0 ::::80      ::::*      LISTEN      2791/apache2
```

If the server were listening on port 443 as well, we would've seen the following output instead:

```
| tcp6      0      0 ::::80      ::::*      LISTEN      3257/apache2
| tcp6      0      0 ::::443     ::::*      LISTEN      3257/apache2
```

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 SSL. 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've already gone 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 in to port 443 that hasn't been identified in any other virtual host. In addition, the `SSLEngine on` option enables SSL traffic. Right after that, we have options for our SSL 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 being applied to the `SSLOptions +StdEnvVars` setting, which enables default environment variables for use with SSL. 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 SSL, our site isn't secure just yet. We'll need SSL 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.

As we go through this process, I'll first walk you through setting up SSL 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 -c
```

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 file names 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 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 SSL certificates that were signed by a certificate authority. The process for installing signed SSL 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 SSL 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 SSL is requested. We can add a `ServerName` option to that file to ensure our site supports SSL.

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 in to 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 the `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 the `ServerName` to the domain that corresponds to your site. Here's an example virtual host file for a hypothetical site that uses SSL. 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 the `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 them their 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

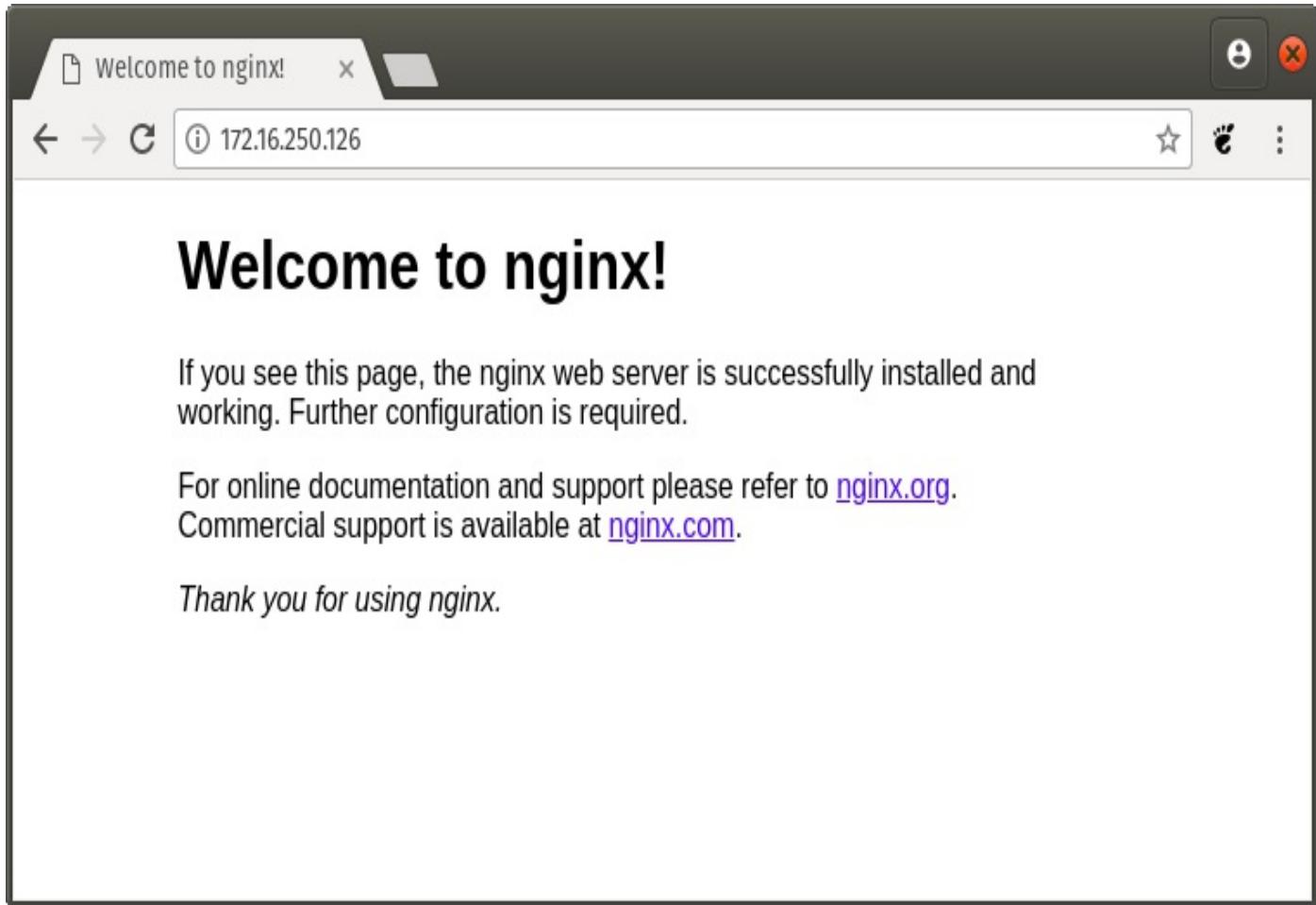
Apache isn't the only technology that is capable of allowing you to host web content on your server. NGINX also serves the same purpose and is gaining popularity quite rapidly. Although Apache is still the most common right now and is what I recommend for hosting sites, it's a good idea to at least be familiar with NGINX and learn its basics.

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 in 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. This time, NGINX's version instead of the one that ships with Apache. It certainly looks boring in comparison, but it works:



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.

In case 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. Change 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 SSL? 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 in which 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 SSL 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.

Setting up failover with keepalived

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 master server isn't responding. This allows you to run a service on multiple servers, with a server taking over in the event 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 master server will have ownership of the floating IP, and other servers will continually check the availability of the master, claiming the floating IP for themselves whenever the current master 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 in order 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 the **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 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 master 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.

```
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.

The `virtual_router_id` is a very special number that can be anything from 0-255. 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 their 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 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 the `priority`. This number must absolutely be different on each server. The `keepalived` instance with the highest `priority` is always considered the master. 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 master has the highest priority, you're in good shape.

```
|   virtual_ipaddress {  
|     192.168.1.200  
|   }
```

In the final block, we're setting the virtual IP address. 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 master 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 -l 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 slave server to a lower number than what you used on the master server
- The `virtual_ipaddress` should be the same on both
- The `virtual_router_id` should be the same on both
- Start or restart `keepalived` on your slave 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 inside 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 master 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 master. 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 slave server, you should also see that the slave is now holding your floating IP address. Neat, isn't it? To see your master reclaim this IP, start the `keepalived` daemon on your master server and wait a few seconds.

Congratulations, you set up a cluster for your Apache implementation. If the master server encounters an issue and drops off the network, your slave server will take over.

As soon as you fix your master 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. Personally, I use it every day and can't imagine life without it. 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 9, Managing Databases](#). I'll give you all the commands you'll need to set up the database in this section, but refer back to [Chapter 9, Managing Databases](#) 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 this writing, the download link is at the top-right corner of the page. Once there, look for a Get Nextcloud Server section, and click on the Download button right underneath that. You should see a large Download Nextcloud button here, 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.

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-13.0.0.zip
```

This command will download the Nextcloud software locally to your current working

directory. Next, we'll need to `unzip` the archive:

```
| unzip nextcloud-13.0.0.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>
```

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-php7.2` 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-php7.2 php7.2-curl php7.2-gd php7.2-intl php7.2-mbstring php7.2-m
```

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 9](#), 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.

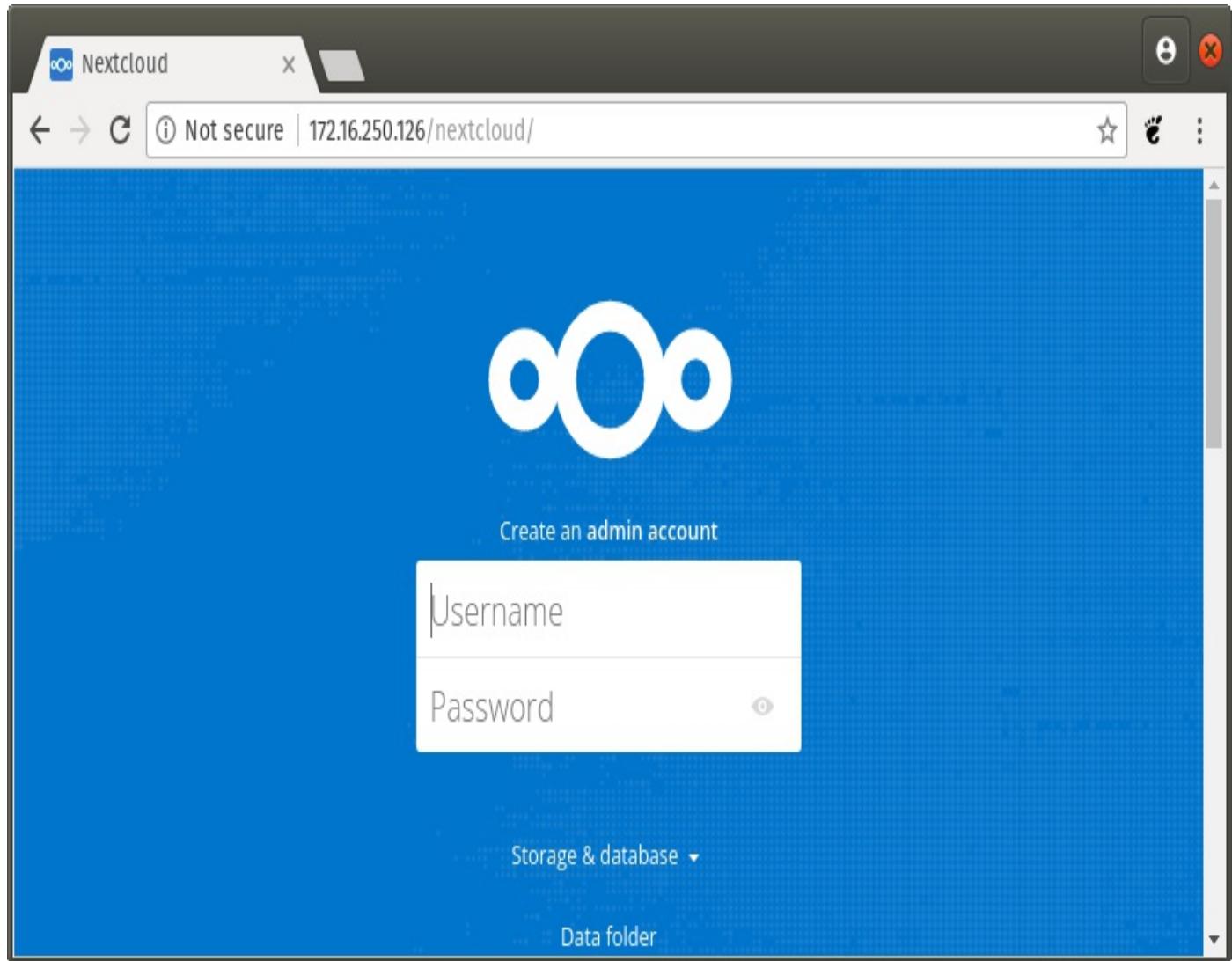
Now we have all we need in order to configure Nextcloud. You should now be able to visit your Nextcloud instance in a web browser. Just enter a URL similar to the following, replacing the sample IP address with the one for your server:

```
| http://192.168.1.100/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:



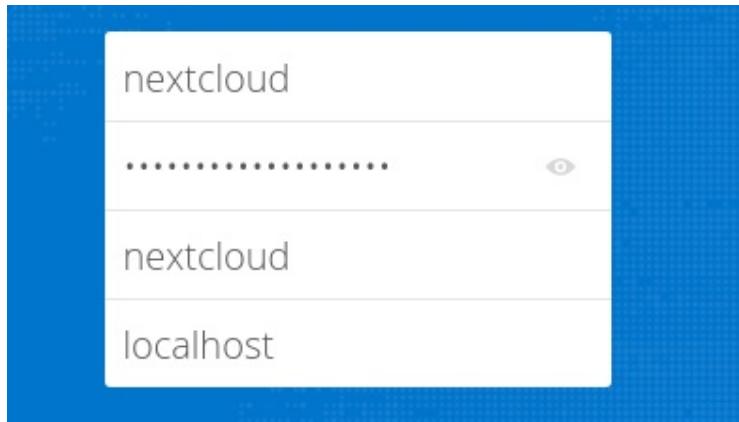
If you do not see this page, make sure that the `/var/www/html/nextcloud` directory is accessible via Apache. Also, make sure you have an appropriate virtual host for Nextcloud referencing this directory as its Document Root.

This page will ask you for several pieces of information. First, you'll be asked for 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.

The `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:



Nextcloud configuration page (database section)

The 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 the Database name. The password will be whatever it is you used for the password when we set up the database user account and grant. 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 server's IP address here.

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 have check out your Nextcloud server. To do so, click on the top-right corner of the Nextcloud page, where it shows the username you're logged in with. When you click on this, you'll see an option for Users. On the Users screen, you'll be able to add additional users to access Nextcloud. Simply fill out the Username and Password fields at the top of the screen, and click on Create.

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 Files and Gallery plugins. There are many more apps that you can enable in order to extend its functionality. On 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 Calendar and Contacts, Tasks.

Now, you have your very own Nextcloud server. I find Nextcloud to be a very useful platform, something I've come to depend on for my calendar, contacts, and more. 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 desktop or laptop. 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 SSL. With Apache, we can use self-signed certificates, or we can purchase SSL 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 master 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 learn more advanced command-line techniques

Questions

1. What are two popular solutions for serving web pages in Ubuntu Server?
2. What is the name of Apache's background daemon that you can manage with `systemctl`?
3. Apache features ____, which allow you to serve multiple web pages from one server.
4. The ____ command allows you to enable a site configuration in Apache.
5. ____ certificates allow you to benefit from secure connections, but browsers don't trust them by default.
6. The ____ is the default directory where website files are served from.
7. Which port is generally reserved for secure HTTPS connections?
8. `keepalived` will prefer the server with the highest ____.

Further reading

- **NGINX documentation from the Ubuntu community wiki:** <https://help.ubuntu.com/community/Nginx>
- **Nextcloud 13 Administration Manual:** https://docs.nextcloud.com/server/13/admin_manual/

Learning Advanced Shell Techniques

Throughout this book so far, we've been using the command line quite heavily. We've installed packages, created users, edited configuration files, and more using the shell. This time around, we dedicate an entire chapter to the shell so we can become 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:

- Understanding the Linux shell
- Understanding Bash history
- Learning some useful command-line tricks
- Redirecting output
- Understanding variables
- Writing simple scripts
- Putting it all together: Writing an rsync backup script

Understanding the Linux shell

When it comes to the Linux shell, it's important to understand what exactly this 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, such as **Zsh**, **Fish**, **ksh**, and so on. Bash is the default command shell for pretty much every Linux distribution in existence. It's possible there's a Linux distribution I have yet to try that defaults to another shell, but Bash is definitely the most common. In fact, it's even available on macOS in its Terminal, and in Windows by installing the Ubuntu plugin. Therefore, by understanding the basics of Bash, your knowledge will be compatible with other distributions and platform. 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, 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:

```
| cat /etc/passwd
```

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 this user to open a shell to do other things. The `/usr/sbin/nologin` shell also doesn't allow the user to log in, but will provide a polite message letting them know.

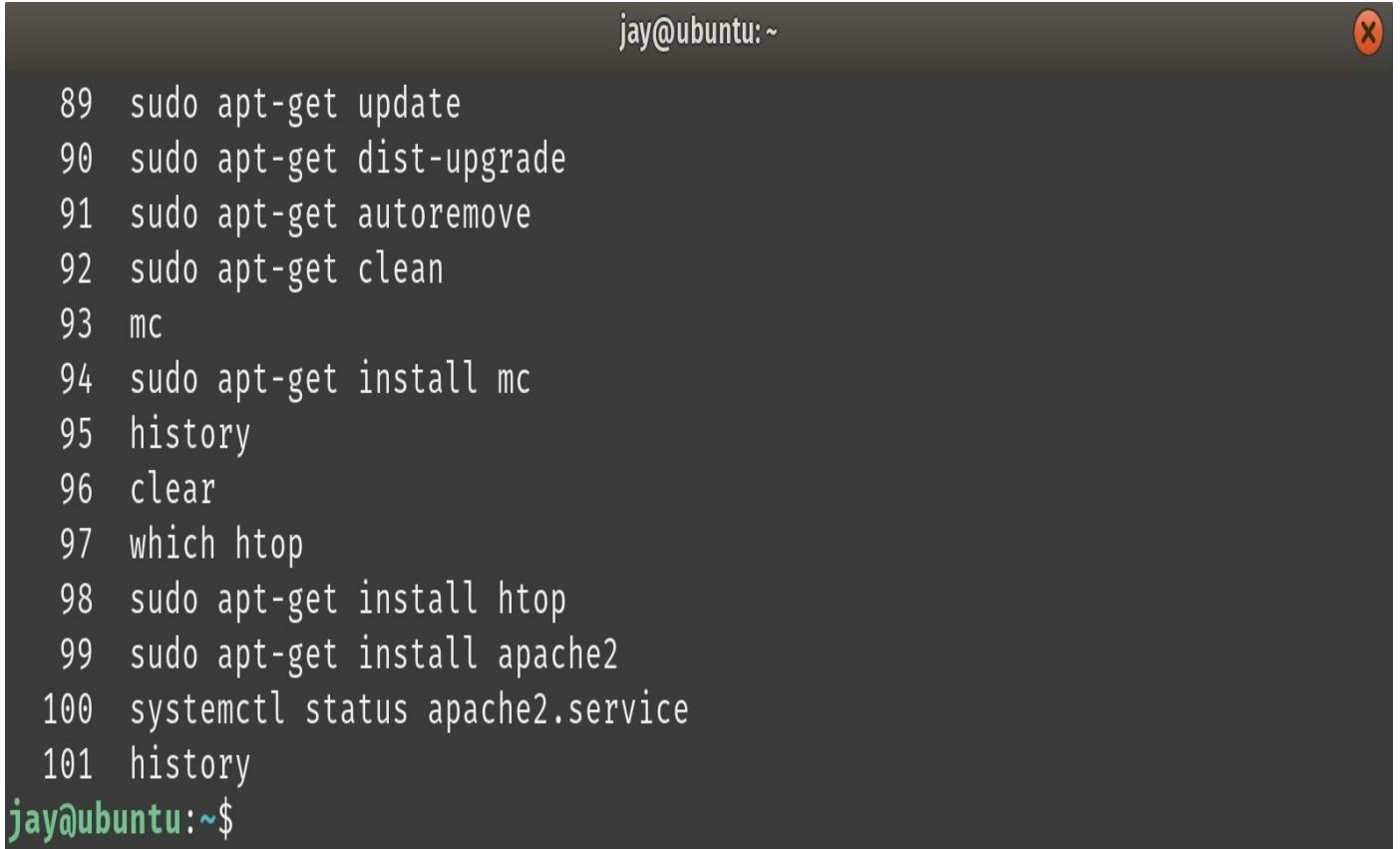
The shell program itself is responsible for reading the commands you type and having the Linux kernel execute them. Some shells, Bash notably, have additional features, such as history, that are very useful to administrators.

Understanding Bash history

Speaking of history, let's dive right into that concept. Bash keeps track of all the commands you enter, 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 unless you have a spiteful user that is also sloppy.

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, you know now. 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 you can also simply type `history` in the shell and see a list:



The screenshot shows a terminal window with a dark background. At the top, the prompt `jay@ubuntu:~` is visible. On the right side of the window, there is a small red circular icon with a white 'X' inside. The main area of the terminal displays a list of command history entries, each preceded by a line number (e.g., 89, 90, 91, etc.). The commands listed are:

```
89 sudo apt-get update
90 sudo apt-get dist-upgrade
91 sudo apt-get autoremove
92 sudo apt-get clean
93 mc
94 sudo apt-get install mc
95 history
96 clear
97 which htop
98 sudo apt-get install htop
99 sudo apt-get install apache2
100 systemctl status apache2.service
101 history
```

At the bottom of the terminal window, the prompt `jay@ubuntu:~$` is shown again.

Output from the hostname command

At this point, you can copy and paste a command used previously from this list. Actually, 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 `100` is where I checked the status of the `apache2` service. If I wanted to check that again, I can simply enter the following command:

```
| !100
```

In this case, I typed just four characters, and I was able to recall the previously used command, which was the exact same as typing this:

```
| systemctl status apache2
```

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 100
```

In this case, we can replace `100` with whatever the number is for the item we would want to delete. You may be wondering, ""why delete something from the history?"" and 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/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 do this. 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.

Learning some useful command-line tricks

Productivity hacks utilizing the shell are one 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 it should have. In this section, in no particular order, I'll go over a few tricks that increased my work-flow.

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 16 years and I still forget to use `sudo` from time to time. When we forget `sudo`, we can 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 retype that command.

Going a bit further into 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, which will open the command 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.

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, where 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:

```
| 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 (which we'll get into in more detail later).

Now, it's time for my favorite time-consuming 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 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 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 ten CPU consuming processes:

```
| alias cpu10='ps auxf | sort -nr -k 3 | head -10'
```

- View the top ten RAM consuming processes:

```
| alias mem10='ps auxf | 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 auxf | sort -nr -k 3 | head -10'  
| alias mem10='ps auxf | 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 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. Here's a bonus trick:

```
| cd -
```

That simple command changes your working directory back to the previous directory you were in.

You're welcome!

Redirecting output

Before we go any further, we should have a discussion about redirecting output. One of the many great things about Linux is that it's made up of many utilities, each doing one thing and doing that one thing well. These smaller commands and utilities become even more efficient because we can take the output of one command and redirect it into the input of another command. Take this command as an example:

```
| cat /var/log/syslog | grep apache2
```

Here, we're using the `cat` command to print the contents of the system log stored at `/var/log/syslog`. Using the pipe symbol, we're redirecting the output of that command and feeding it into `grep`, which is looking for the string `apache2`. When used together, we're able to fetch lines from the system log that reference `apache2`. This is useful if we're troubleshooting some sort of issue with the `apache2` service, and we'd like to look at the log file but ignore any lines that aren't relevant to our investigation. This is something you'll probably find yourself doing a lot.

Working with redirecting output is probably best explained with simple text examples. These are admittedly contrived, but understanding how to work with text is important:

```
| echo "this is a test" >> ~/myfile.txt
```

Here, we're simply using the `echo` command, which normally outputs to the Terminal, but instead telling it to direct its output to a file named `myfile.txt`, stored in your home directory. If we run this command over and over, it will continue to append this string to the file. With a single greater-than symbol, the output from `echo` will actually wipe out the contents of `myfile.txt`:

```
| echo "this is a test" > ~/myfile.txt
```

This distinction is important, because you may want to add a new line to the end of a file, and not have the existing contents deleted. Therefore, it's best to get in the habit of using two greater-than symbols by default. There will be, of course, situations in which you do want to wipe out a file, but it's best to make two greater-than symbols the one you commit to muscle memory.

Effectively, we're working with a **File Descriptor** called **Standard Output** (also

referred to as **stdout**) with commands that produce output. File descriptors are numbers that refer to open files, and these numbers are integers. Standard output is a special file descriptor, with an integer of `1`. Anytime text is printed to the Terminal, we call that standard output. Therefore, when the `ls` command is run, it's printing the contents of the current working directory to standard output. Another important file descriptor is **Standard Error (stderr)**, which constitutes output that is considered to be produced in error. Standard error is designated by a file descriptor of `2`. Why is this important? Sometimes when working with the shell, we would like it to do something different with error output than with normal output. Perhaps we want to run a command, and have all error output go to a file, but any non-error output to not be captured. Sometimes, even vice versa. The best way to illustrate this is with an example. Let's run this command (make sure to NOT use `sudo`):

```
| find /etc -name *apache*
```

With this hypothetical example, we're telling the shell to look in the `/etc` directory and return a list of files and/or directories that contain `apache` in the name.

The problem is that as a normal user (we didn't use `sudo`) we don't have privileges to access every file and directory in `/etc`. Among the output of this variation of the `find` command, we might see lines of output similar to this:

```
find: '/etc/lvm/backup': Permission denied
find: '/etc/lvm/archive': Permission denied
find: '/etc/vpnc': Permission denied
find: '/etc/ssl/private': Permission denied
find: '/etc/libvirt/secrets': Permission denied
find: '/etc/polkit-1/localauthority': Permission denied
```

With really large directories, we can have so many errors that we won't even be able to see the results that we actually care about. Obviously, you could just prefix the `find` command with `sudo`, but we don't want to use `sudo` unless we really need to. For our purposes, there aren't going to be any configuration files for Apache that we don't at least have read access to, so we don't need to use `sudo` here, but trimming the output would be helpful. We can do this instead:

```
| find /etc -name *apache* 2>/dev/null
```

We've redirected output with a greater-than symbol before, but here we're doing it a bit differently. We're directing output from the `find` command with `2>` instead of `>`. Since `2` is the identifier of `stderr`, we're telling the shell to direct any errors to `/dev/null`. `/dev/null` is a special device that is similar to a black hole: anything that enters it is never seen or

heard from again. Basically, anything copied or moved to `/dev/null` is eliminated and wiped from existence. Any output we direct there isn't printed, but deleted. In fact, don't do this, but you can move a file there to get rid of it too (it's still easier to use the `rm` command though).

Going further, we can also direct specific output to a file as well. Here's an example:

```
| find etc -name *apache* 1> ~/myfile.txt
```

Here, we're copying `stdout` (not `stderr`) to a file. The ability to copy `stdout` or `stderr` to specific places will increasingly become useful as you become more proficient with the shell.

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 far as what separates a scripting language from a programming language, and that line is blurred more and more as new languages come out. As with any scripting language, Bash supports variables. The concept of variables is very easy 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 dollar symbol, but you do when you retrieve it.



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 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! That's 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 server). Any of these variables can be accessed at any time, and may even prove beneficial in scripts.

We've already gone over standard output and standard error, but there's also another we haven't gone over yet, and that is **Standard Input (stdin)**. Basically, `stdin` is just a fancy way of describing the process of taking input from a user and storing it in a variable. Try this command:

```
| 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, 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're getting a bit ahead of ourselves here, but I wanted to at least make sure you understand the concept of standard input, since it directly relates to setting variables as we're discussing here.

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 into a script. This is a great habit to get into.

A script is a very simple concept. A script is 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 the simpler `nano`. Since we've edited text files before, I'm under the assumption that you already know how to create and edit files. If not, all you really need to know is that `nano` is a good text editor to begin with and is simple to use.

 There's a big chance that at some point in your Linux career, you'll gravitate toward `vim`; almost everyone does (unless you're one of those people that use `emacs`, for some reason). If in doubt, just stick with `nano` for now.

Basic usage of `nano` is really easy: all you need to do is give the `nano` command a path and filename. You don't actually have to give it a filename, but I find it easier to do so. After you've typed some content, you can press `Ctrl + O` to save and then `Ctrl + X` to exit the editor. To get started, let's create a simple script so we can see what the process looks like:

| `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"
```

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"
```

The first line 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` we see on the 1st line, this is known as a **hash bang**, or **shebang**. Basically, it just tells the shell which interpreter to use 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 apache2  
| sudo apt install libapache2-mod-php7.2  
| sudo a2enmod php  
| sudo systemctl restart apache2
```

We haven't gone over any advanced shell concepts yet, so this will do for now. But what we've done 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.

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-php7.2
    sudo a2enmod php
    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. We use the `-y` option here, because in a script we probably don't want to have it wait for confirmation. Normally, I don't like using the `-y` option, but in a script it's perfectly acceptable.

Finally, we close out our `if` statement with the word `if` backwards (`fi`). If you forgot to do this, the script will fail.

In 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. We just set a number (integer) here, so we would've only used quotation marks if we wanted to set 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 if 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 do something repeatedly until some 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**. If you used `-ge 0` instead, you would've created exactly that.

```
| 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 incrementer 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. We included four turtle names in our list.

Feel free to practice the concepts I listed here until you get the hang of everything. If you want to explore further, I have an entire series about Bash scripting on my YouTube channel you can use to expand your knowledge. Check out the Further reading section for the URL.

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 further 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/08-17-2018 /src /target
```

This variation of `rsync` is the exact one we used back in [Chapter 8](#), Sharing and Transferring Files. Feel free to consult that chapter for an overview of how this utility works. This example `rsync` command uses the `-a` (archive) option, which we've discussed before. Basically, it retains the metadata of the file, such as the time stamp 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 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've been renamed in this way (or deleted), to instead just copy it to another directory. In this case, we send any files that would've been overwritten to the `/backup/incremental/08-16-2018` 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 the directory we're using for the `--backup-dir`. The date changes every day, so we shouldn't be hard-coding 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 parenthesis 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 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
```

Now, simply create a cron job for this script to be run periodically. Refer to [Chapter 6](#), Controlling and Monitoring Processes for more information on how to create cron jobs if you need a refresher on that.

Summary

In this chapter, we dived in to some 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 16 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 virtualization, which is one of my favorite subjects. We'll set up our very own virtualization server, which will be a lot of fun. See you there!

Questions

1. What are some other command shells, aside from Bash?
2. An _____ allows you to create a new command that executes a different command or string of commands.
3. A _____ should be the first line in your script and informs the shell which interpreter to use.
4. Commands that you've executed previously can be seen by using the _____ command.
5. _____ captures output from commands that are run and is identified by the number 1.
6. Error messages are written to _____ and can be identified by the number 2.
7. The _____ operator redirects output to a file and appends it to the end of that file.
8. The _____ operator redirects output to a file and wipes the content of that file.
9. After storing information in a variable, you must include the _____ symbol in the variable name in order to fetch its contents.

Further reading

- **Comparison operators for Bash:** <http://tldp.org/LDP/abs/html/comparison-ops.html>
- **Introduction to Bash scripting tutorial series (YouTube):** <https://tinyurl.com/mfeaqx8>
- **Commandlinefu:** <https://www.commandlinefu.com/commands/browse>

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 itself 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 or hypervisor. Since Ubuntu features the latest advancements of the Linux kernel, virtualization is actually built right into it. After installing just a few packages, 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:

- Setting up a virtual machine server
- Creating virtual machines
- Bridging the virtual machine network
- Simplifying VM creation with cloning
- Managing virtual machines via the command line

Setting up a virtual machine server

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 running in a solution such as VirtualBox, Parallels, VMware, or one of the others. In this section, we'll see how to use an Ubuntu Server in place of those solutions. While there's certainly nothing wrong with solutions such as VirtualBox, Ubuntu has virtualization built right in, in the form of a dynamic duo consisting of **Kernel-based Virtual Machine (KVM)** and **Quick Emulator (QEMU)**, which together form a virtualization suite that enables Ubuntu (and Linux in general) to run virtual machines without the need for a third-party solution. KVM is built right into the Linux kernel, and handles the low-level instructions needed to separate tasks between a host and a guest. QEMU takes that a step further and emulates system hardware devices. There's certainly nothing wrong with running VirtualBox on your Ubuntu Server, and many people do. But there's something to be said of a native system, and KVM offers a very fast interface to the Linux kernel to run your virtual machines 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 virtual machines 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 virtual machines without too much of an impact. Depending on what kind of machine you're using, CPU and RAM may present a bottleneck, 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 virtual machine extensions. The performance of virtual machines may suffer without this support, if you can even get VMs to run at all without these 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 virtual machines on in order to find out whether your CPU supports virtualization extensions. A result of `1` or more means

that your CPU does support virtualization extensions. A result of 0 means it does not:

```
| egrep -c '(vmx|svm)' /proc/cpuinfo
```

Even if your CPU does support virtualization extensions, it's usually a feature that's disabled by default with most end-user PCs sold today. 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 called VT-x or AMD-V. Some chipsets may simply call this feature virtualization support or something along those lines. 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.

There are two ways in which you can use and interface with KVM. You can choose to set up virtualization on your desktop or laptop computer, replacing a solution such as VirtualBox. Alternatively, you can set up a server on your network, and manage the VMs running on it remotely. It's really up to you, as neither solution will impact how you follow along with this chapter. Later on, I'll show you how to connect to a local KVM instance as well as a remote one. It's really simple to do, so feel free to set up KVM on whichever machine you prefer. If you have a spare server available, it will likely make a great KVM host. Not all of us have spare servers lying around though, so use what you have.

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 virtual machines on the same machine simultaneously. This probably goes without saying, but I wanted to mention it just in case you didn't already know. You can certainly have both solutions installed on the same machine, you just can't have a VirtualBox VM up and running, and then expect to start up a KVM virtual machine at the same time. 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 virtual machines can take quite a bit of space. The default directory for KVM virtual machine 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 virtual machine disk images on another volume. Or, you can simply create a symbolic link that will point this directory somewhere else. The choice is yours. If your `root` filesystem has at least

10 GB available, you should be able to create at least one virtual machine without needing to configure the storage. I think it's a fair estimate to assume at least 10 GB of hard drive space per virtual machine.

We'll also need to create a group named `kvm` as we're going to allow members of this group to manage virtual machines:

```
| sudo groupadd kvm
```

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-bin qemu-kvm qemu-system
```

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
```

Let's stop this service for now, as we have some additional configuration to do:

```
| sudo systemctl stop libvirtd
```

Next, make the root user and the `kvm` group the owner of the `/var/lib/libvirt/images` directory:

```
| sudo chown root:kvm /var/lib/libvirt/images
```

Let's 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+rwx /var/lib/libvirt/images
```

The primary user account you use on the server should be a member of the `kvm` group. That way, you'll be able to manage virtual machines without switching to `root` first. Make sure you log out and log in again after executing the next command, so the changes take effect:

```
| sudo usermod -aG kvm <user>
```

At this point, we should be clear to 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
```

On your laptop or desktop (or the machine you'll be managing KVM from), you'll need a few additional packages:

```
| sudo apt install ssh-askpass virt-manager
```

The last package we installed with the previous command was `virt-manager`, which is a graphical utility for managing KVM virtual machines. As a Linux-only tool, you won't be able to install it on a Windows or macOS workstation. There is a way to manage VMs via the command line which we'll get to near the end of this chapter, but `virt-manager` is definitely recommended. If all else fails, you can install this utility inside a Linux VM running on your workstation.

We now have all the tools installed that we will need, so all that we need to do is configure the KVM server for our use. There are a few configuration files we'll need to edit. The first is `/etc/libvirt/libvirtd.conf`. There are a number of changes you'll need to make to this file, which I'll outline below. First, you should make a backup copy of this file in case you make a mistake:

```
| sudo cp /etc/libvirt/libvirtd.conf /etc/libvirt/libvirtd.conf.orig
```

Next, look for the following line:

```
| unix_sock_group = "libvirtd"
```

Change previous line to the following:

```
| unix_sock_group = "kvm"
```

Now, find this line:

```
| unix_sock_ro_perms = "0777"
```

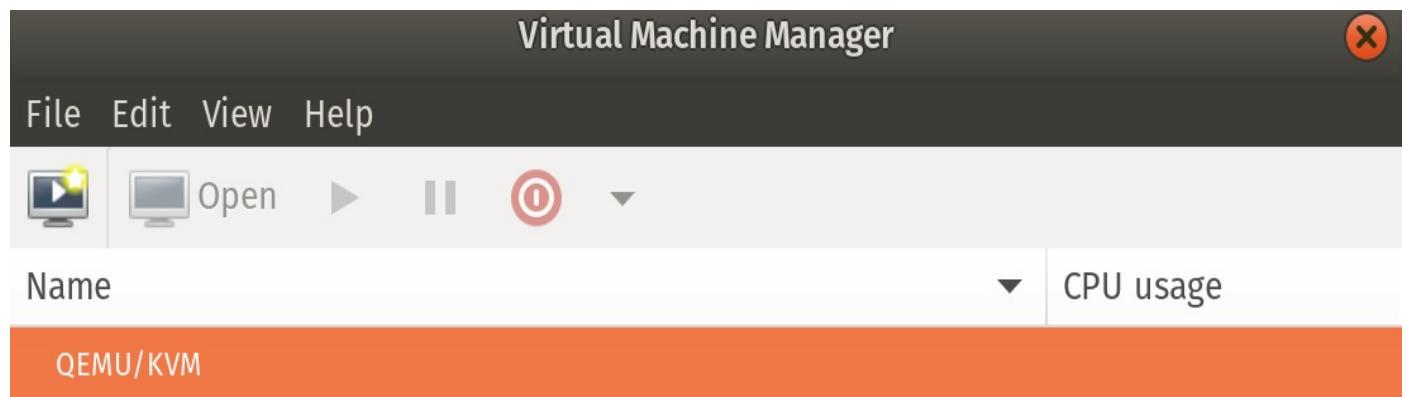
Change it to this:

```
| unix_sock_ro_perms = "0770"
```

Afterwards, restart libvirtd:

```
| sudo systemctl restart libvиртd
```

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 under Virtual Machine Manager. If you have trouble finding it, simply run `virt-manager` at your shell prompt:



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 an external server 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:

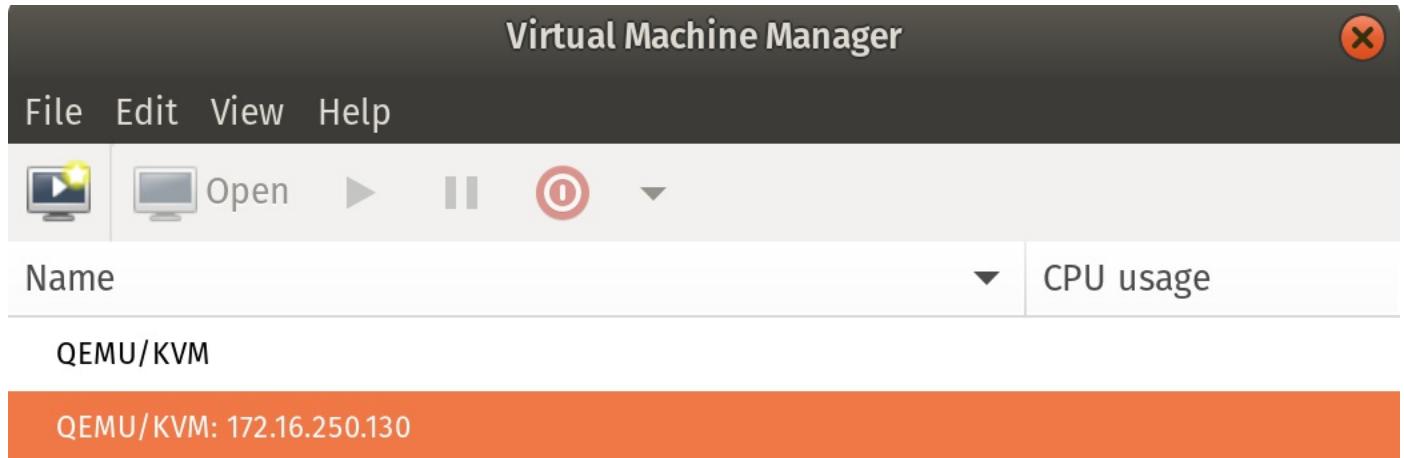


Adding a new connection to virt-manager

In the Add Connection window, you can simply leave the defaults if you're connecting to localhost (meaning your local machine is where you installed the KVM packages). If you installed KVM packages on a remote server, enter the details here. In the screenshot, you can see that I first checked the Connect to remote host box, and then I selected SSH as my connection Method, `jay` for my Username, and `172.16.250.130` was the IP address for the server I installed KVM on. Fill out the details here specific to 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, have permissions to the hypervisor (be a member of the `kvm` group we added earlier) and the `libvirtd` unit 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.

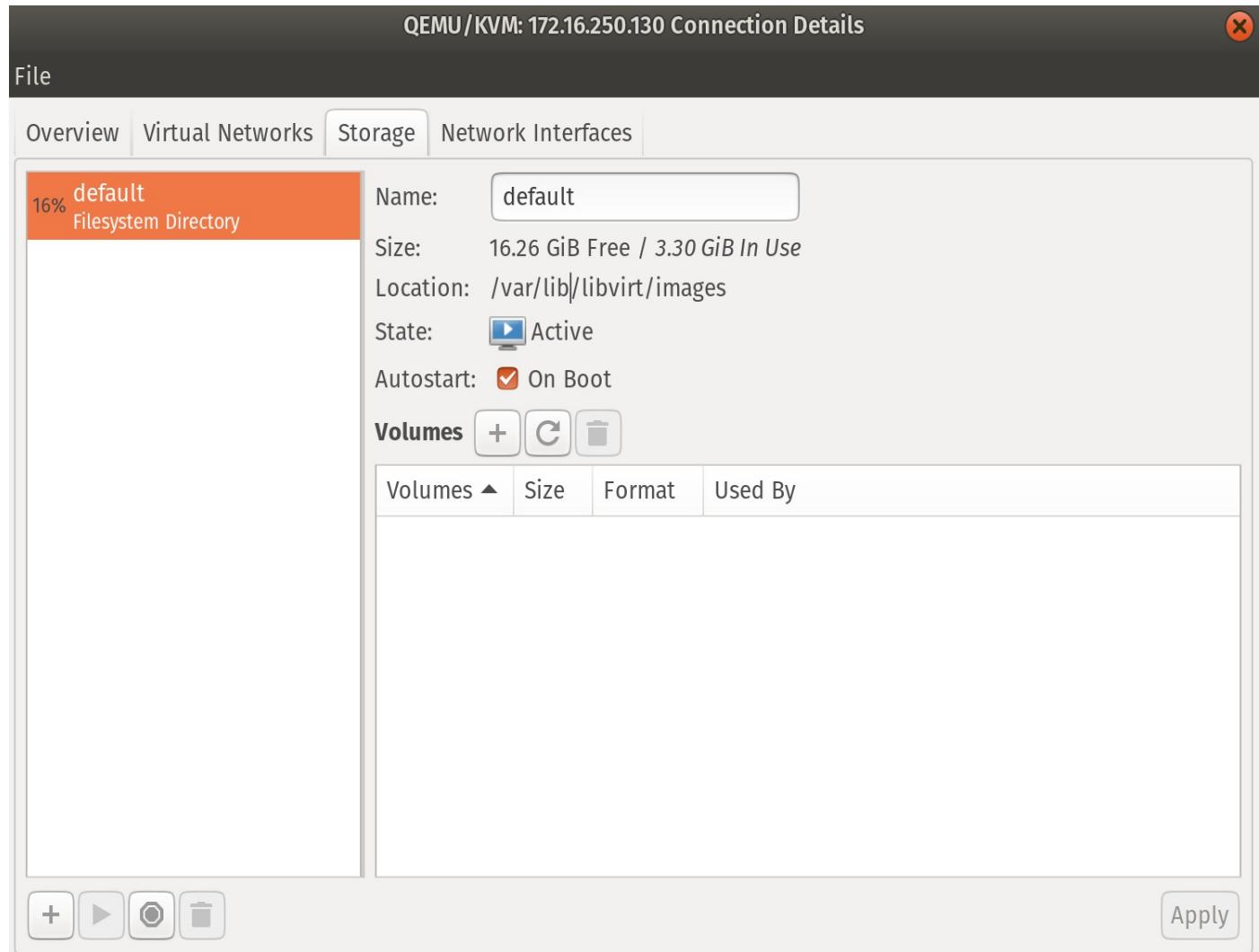
Either way, you should be prompted for your password to your KVM server; type that in and press Enter. You should now have a connection listed in your `virt-manager` application. You can see the connection I added in the following screenshot; it's the second one on the list. The first connection is localhost, since I also have KVM running

on my local laptop in addition to having it installed on a remote server:



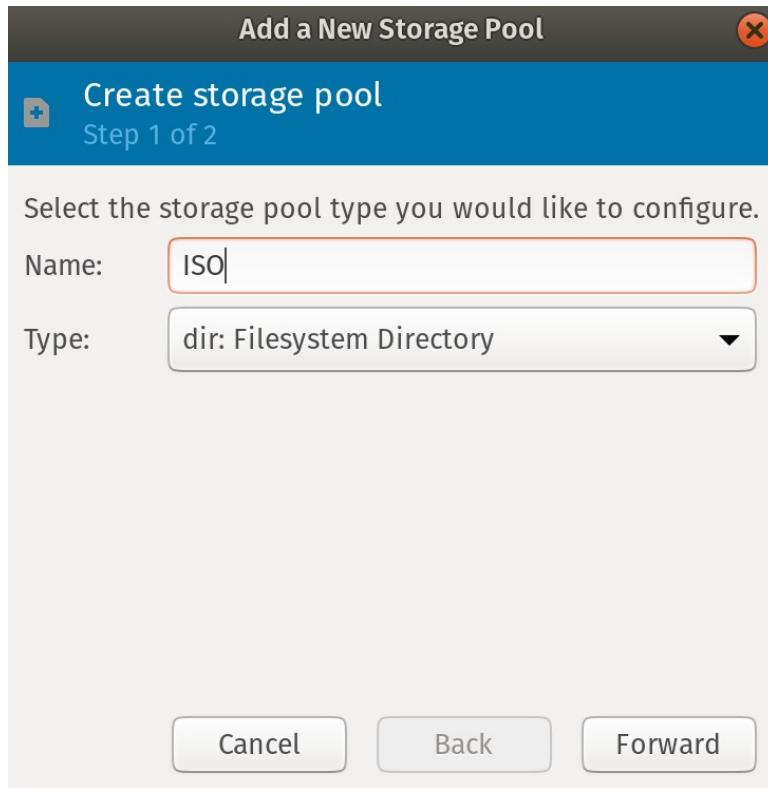
virt-manager with a new connection added

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 virtual machines. When we create a virtual machine, 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 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:



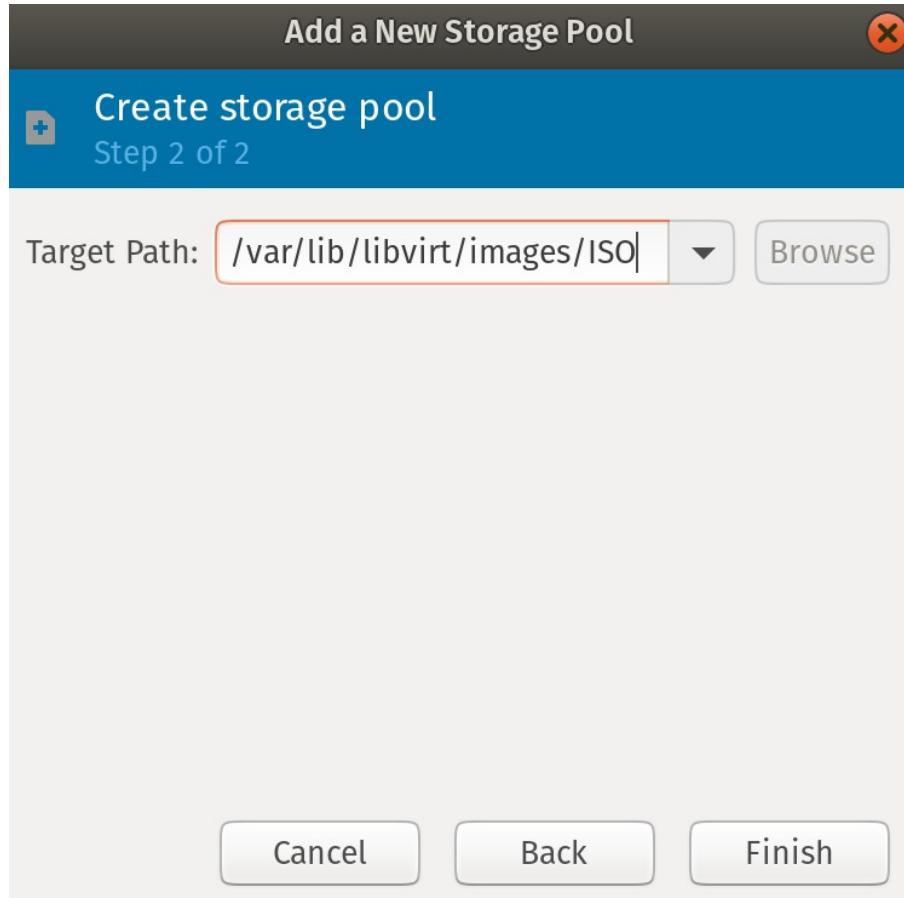
The first screen while setting up a new storage pool

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 to create the new pool:



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. Leave the Type setting as `dir: Filesystem Directory` and click Forward to continue to the next screen:



The second screen while setting up a new storage pool

For the Target Path field, you can leave the default if you want to, which will create a new directory at `/var/lib/libvirt/images/ISO`. If this default path works for you, then you should be all set. Optionally, you can enter a different path here if you prefer to store your ISO images somewhere else. Just make sure the directory exists first. Also, we should update the permissions for this directory:

```
| 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 virtual machines. Our server has a place to store virtual machines 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 download the Ubuntu minimal ISO image from the following wiki article:

<https://help.ubuntu.com/community/Installation/MinimalCD>

The mini ISO file is a special version of Ubuntu that installs only a very small base set of packages and is the smallest download of Ubuntu available.

After you've chosen an ISO file and you've downloaded it, copy it over to your server via scp or rsync. Both of those utilities were covered in [Chapter 8](#), Sharing and Transferring Files. Once the file has been copied over, move the file to your storage directory for ISO images. The default, again, is `/var/lib/libvirt/images/ISO` if you didn't create a custom path.

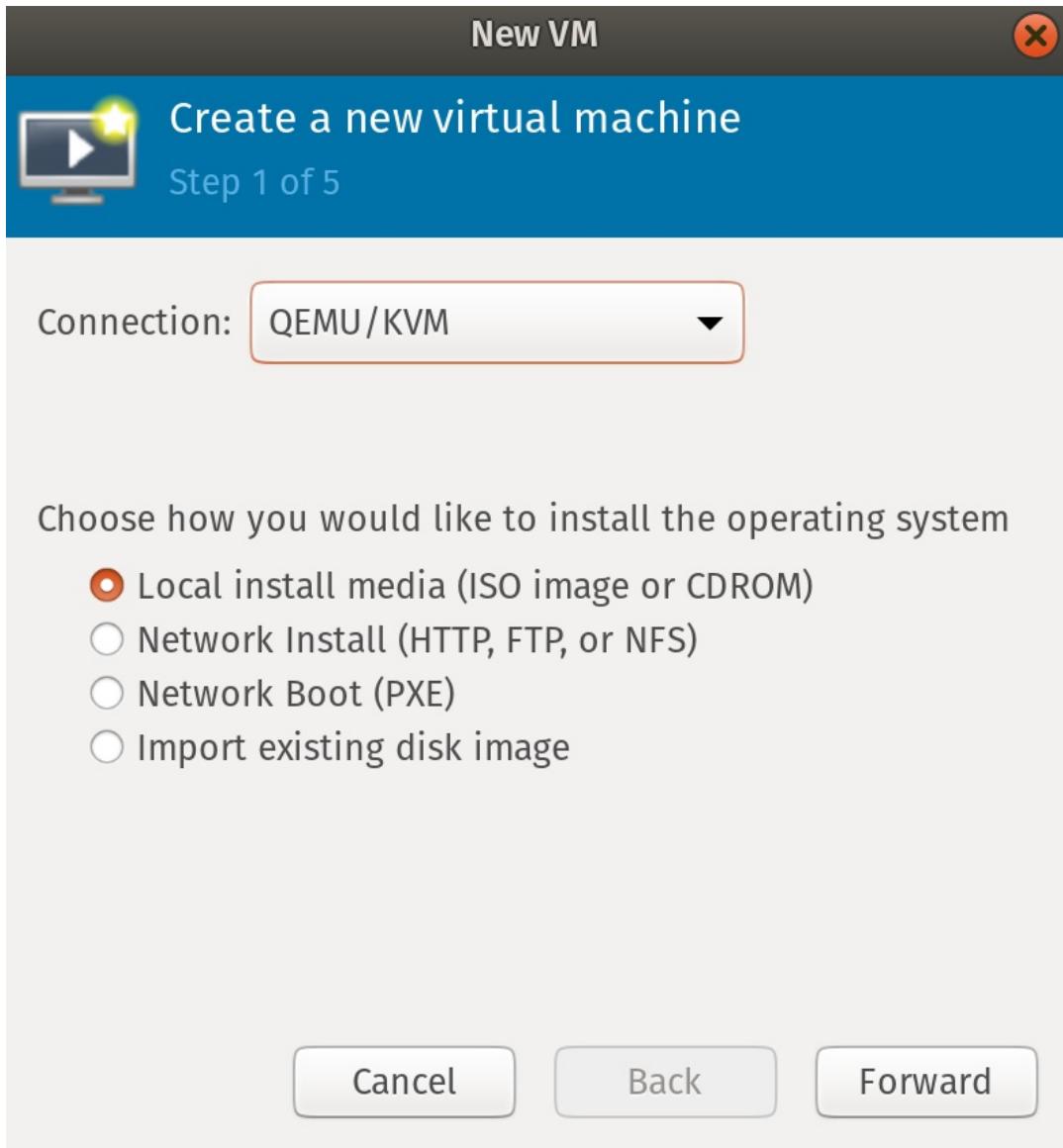
Creating virtual machines

Now, the time has come to put your new virtual machine server to the test and create a virtual machine. 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 at least 512 MB of RAM free
- Go ahead and open up `virt-manager`, and let's get started

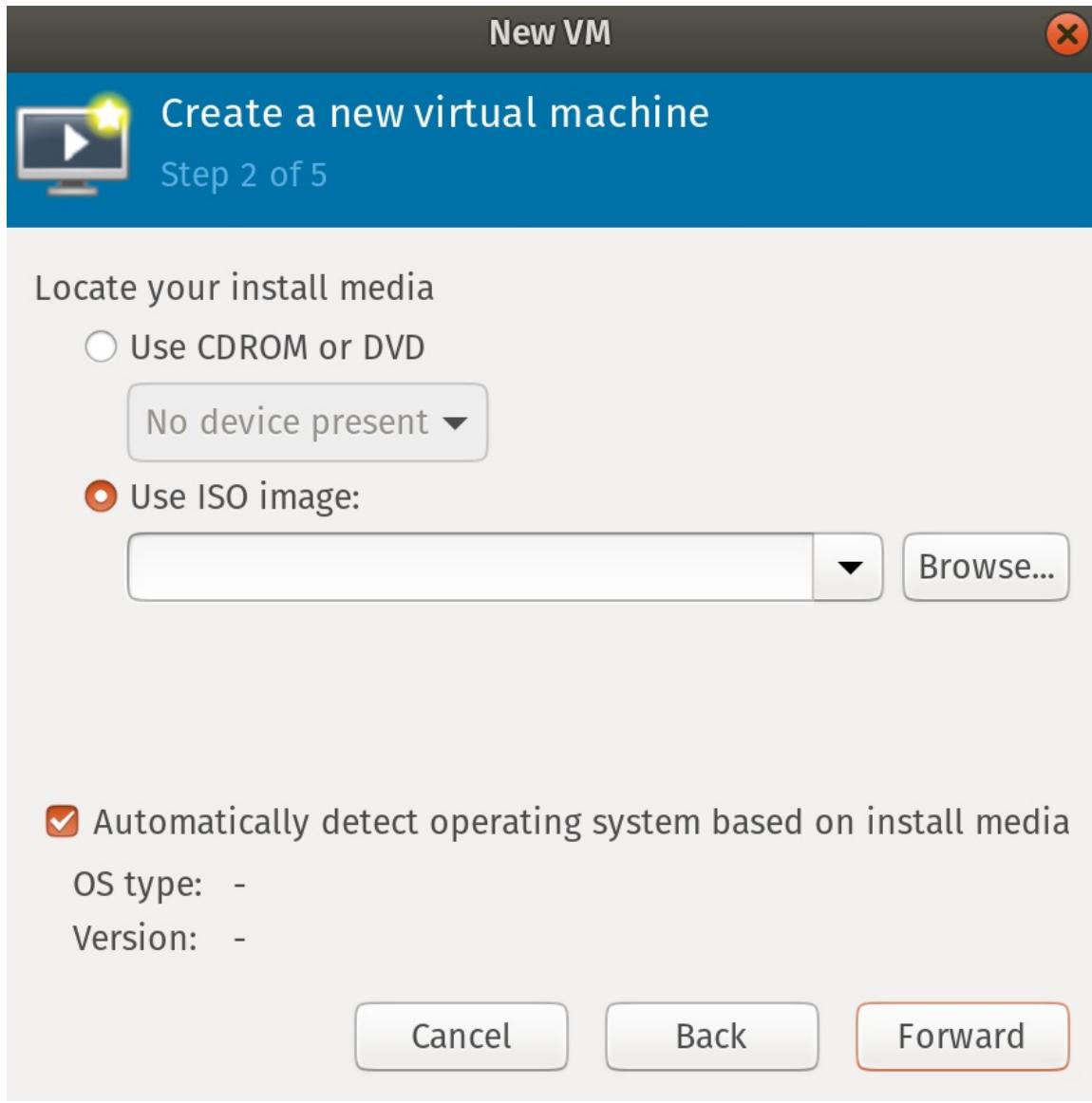
Before continuing, I highly recommend that you set up public key authentication for SSH between your workstation and virtual machine server. If you're using a local connection, you won't need to do this. But when you're connecting to a remote KVM instance, without setting up public key authentication between your workstation and server, you will likely be asked for your SSH password repeatedly. It can be very annoying. If you haven't used public key authentication for SSH yet, please refer back to [Chapter 4](#), Connecting to Networks for an overview.

In `virt-manager`, right-click your server connection and click on New to start the process of creating a new virtual machine:



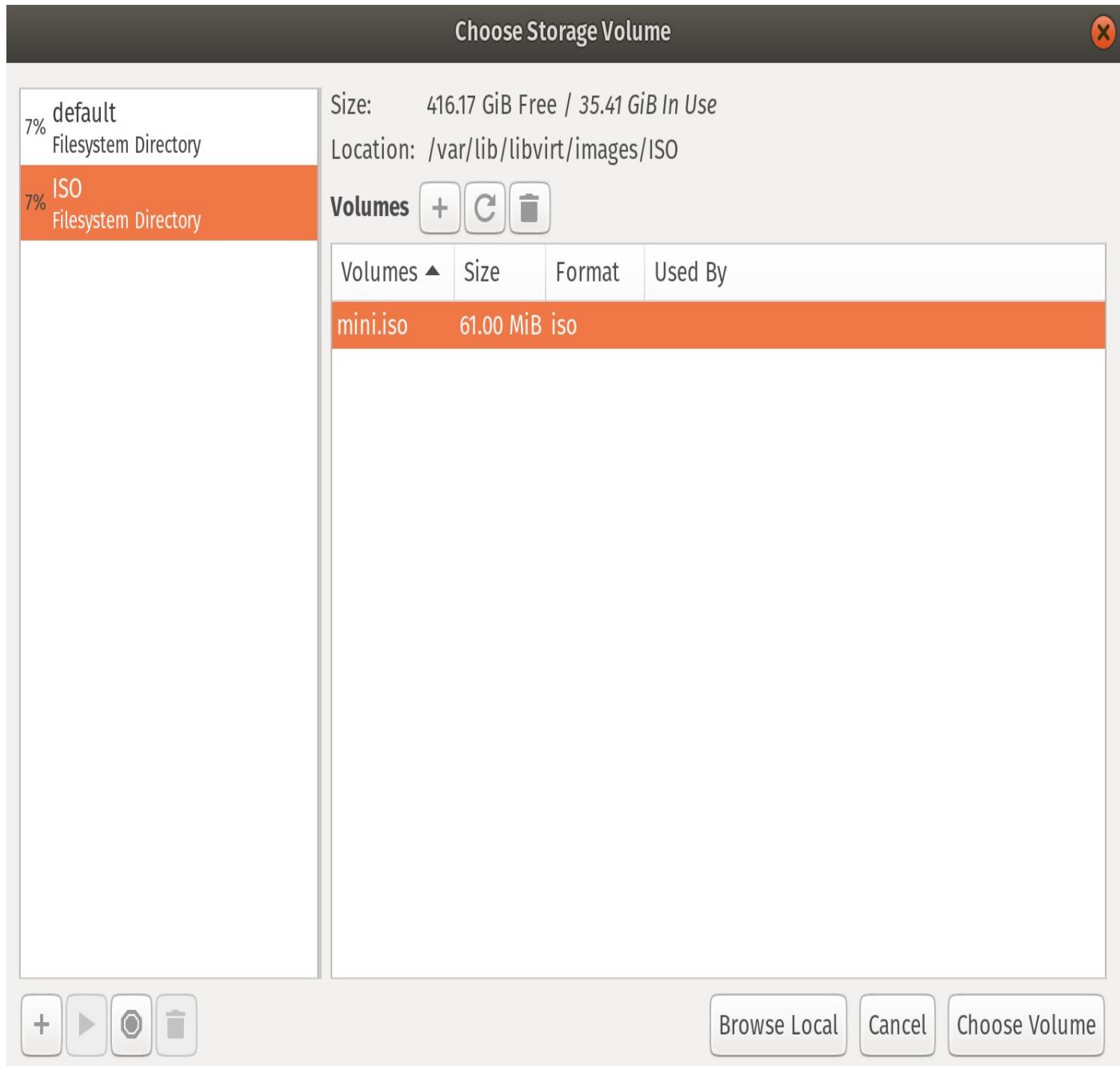
The first screen while setting up a new VM

The default selection will be on Local install media (ISO image or CDROM); leave this selection and click on Forward:



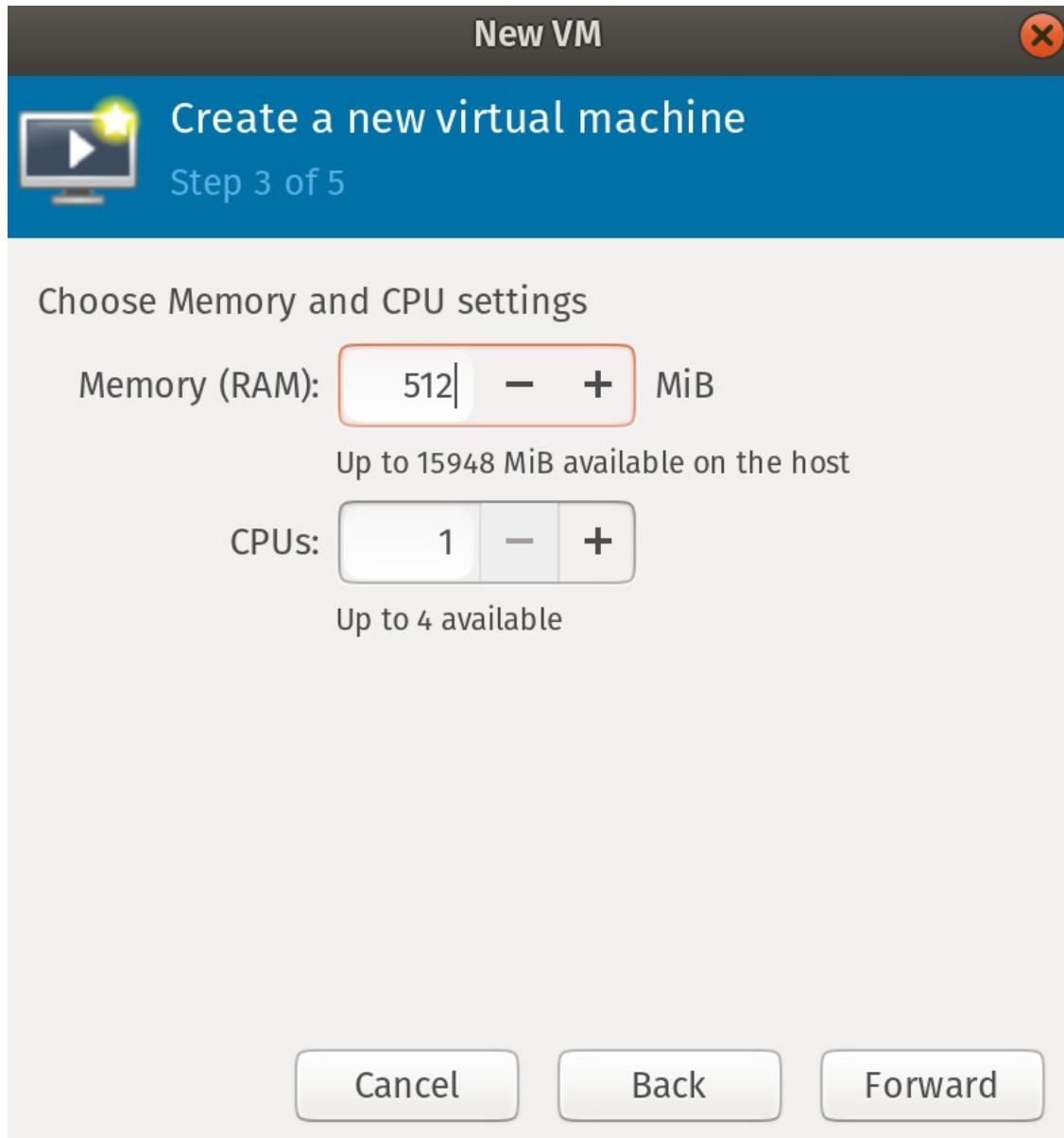
Creating a new VM and setting VM options

On the next screen, click on Browse to open up another window where you can select an ISO image you've downloaded. If you click on your ISO storage pool, you should see a list of ISO images you've downloaded. 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 the minimal version of Ubuntu, because it's small and quick to download. You can use whatever operating system you prefer. Click on Choose Volume to confirm your selection:



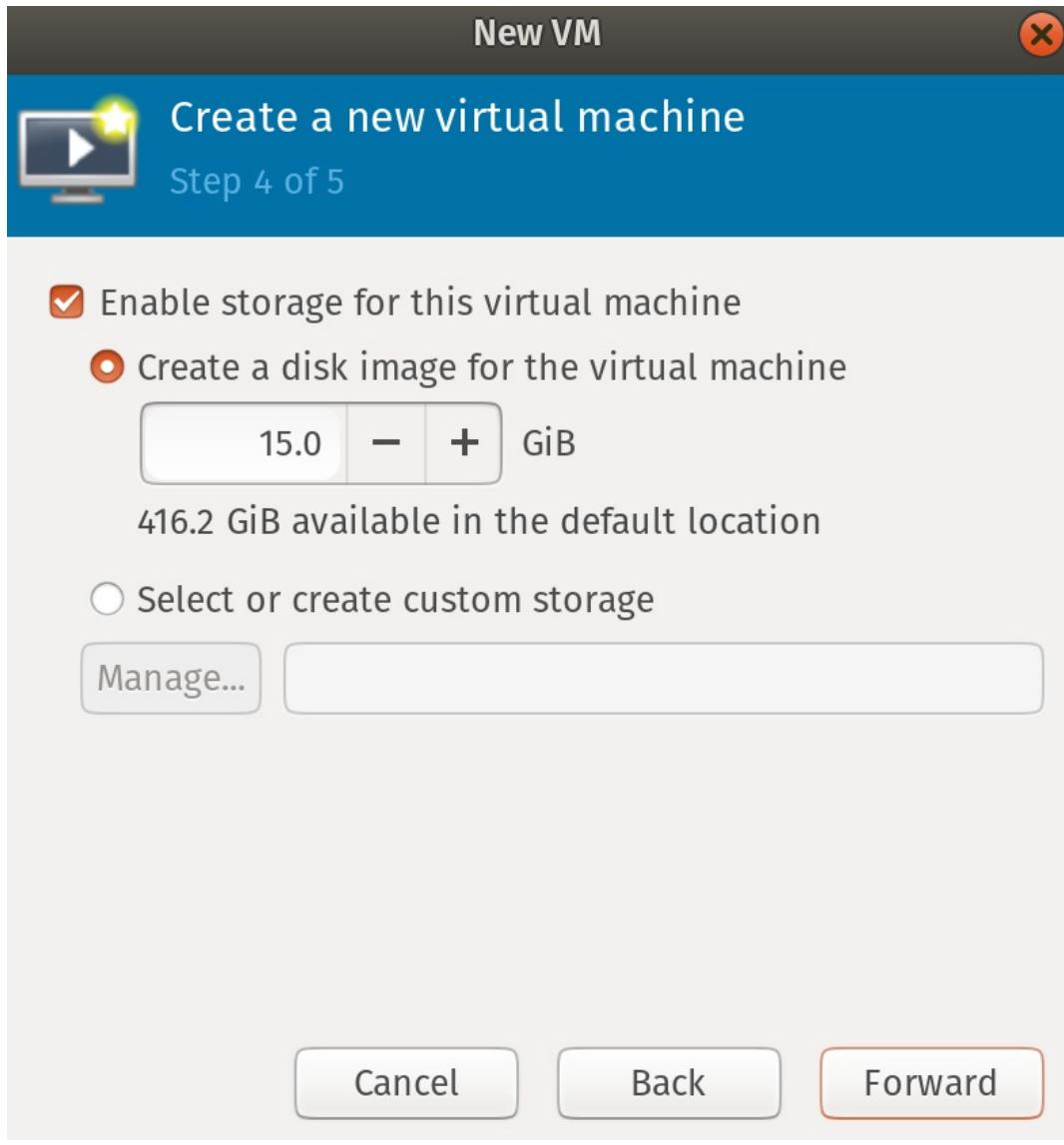
Choosing an ISO image during VM creation

Next, you'll be asked to allocate RAM and CPU resources to the virtual machine. For most Linux distributions with no graphical user interface, 512 MB is plenty (unless your workload demands more). The resources you select here will depend on what you have available on your host. Click on Forward when you've finished allocating resources:



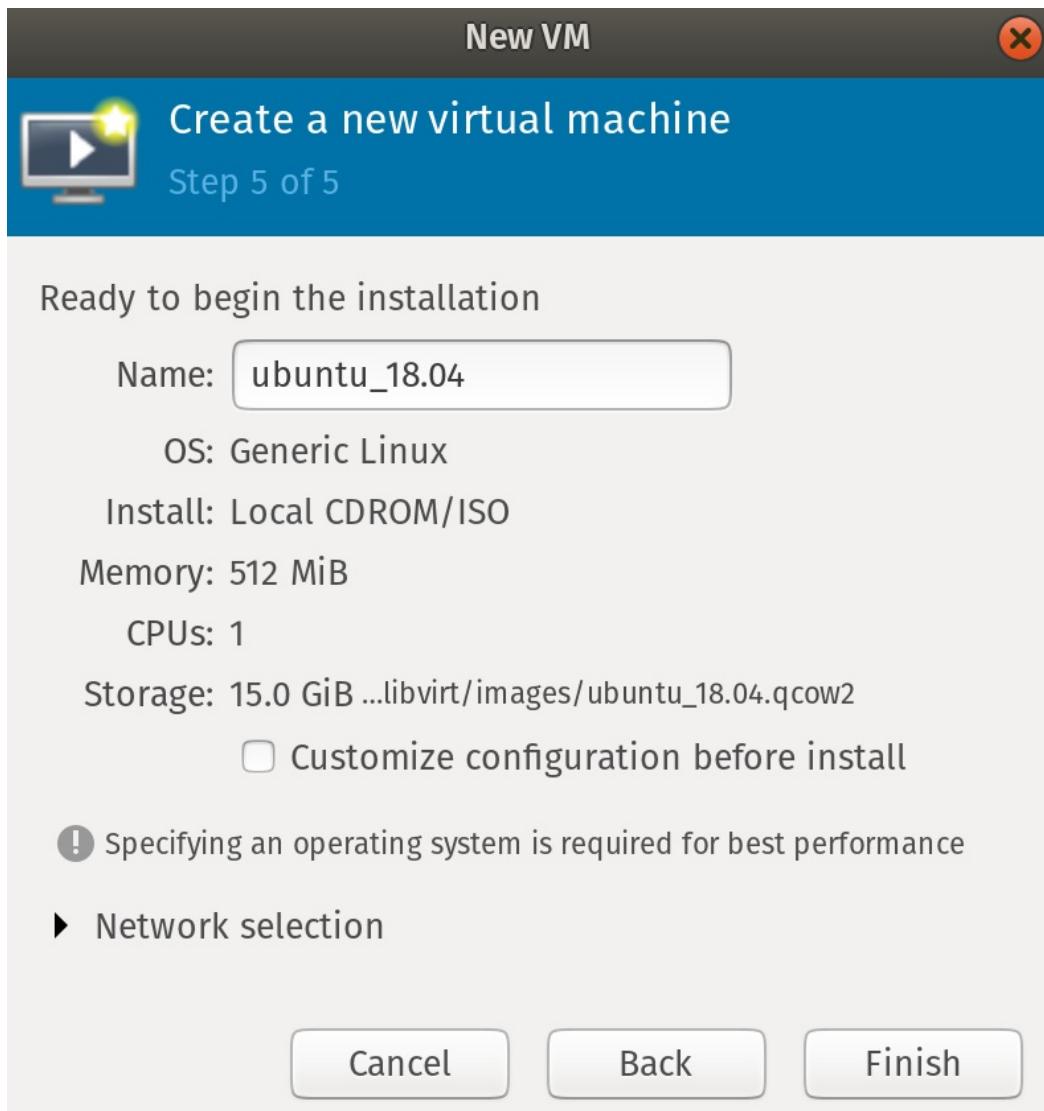
Adjusting RAM and CPU count for the new VM

Next, you'll allocate free disk space for your virtual machine's virtual hard disk. This space won't be used up all at once; by default, KVM utilizes thin provisioning that basically just fills up the virtual disk as your VM needs space. You can select Allocate entire disk now if you'd like to claim all the space all at once, but that isn't necessary. Click on Forward when done:



Allocating storage resources for the new VM

Finally, you'll name your virtual machine. This won't be the hostname of the virtual machine; 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:



Naming the new virtual machine

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.

Bridging the virtual machine network

Your KVM virtual machines will use their own network, unless you configure bridged networking. This means your virtual machines 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 virtual machines by their IP address 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 virtual machine server to power infrastructure for your small office or 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.

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/01-netcfg.yaml` file in your text editor with `sudo`. We already talked about this file in [Chapter 4](#), 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. 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.

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 books code bundle. 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.

```
network:
  version: 2
  renderer: networkd
  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.

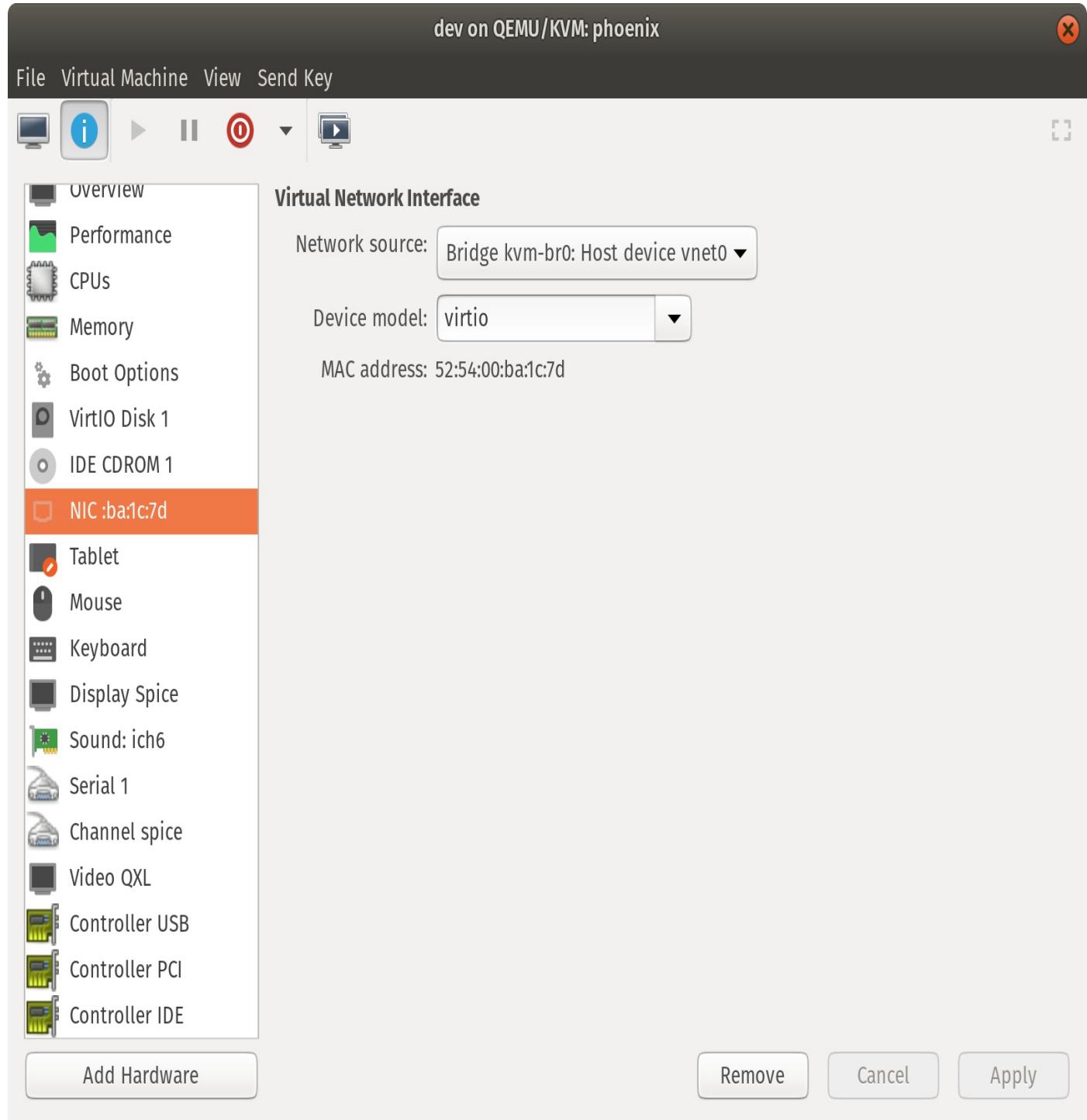
After networking restarts or the server reboots, check whether you can still access network resources, such as pinging websites and accessing other network nodes from it. If you can, you're all set. If you're having any trouble, make sure you edited the `/etc/netplan/01-netcfg.yaml` file properly.

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 onwards, you'll be able to use `br0` for your virtual machine'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 virtual machines.

If you have a virtual machine you've already created that you'd like to switch to utilize

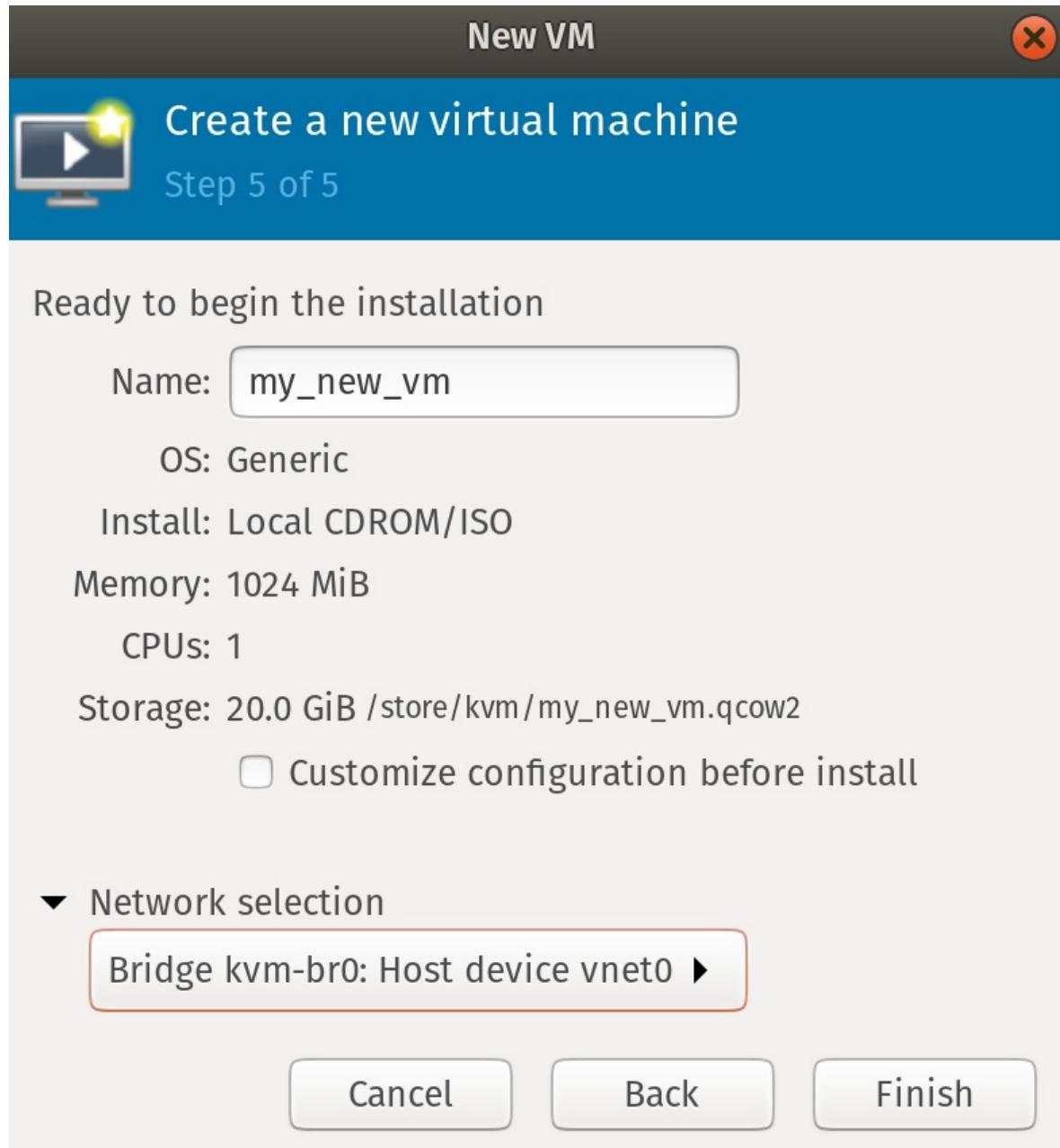
your bridged networking, you can use the following steps to convert it:

1. First, open `virt-manager` and double-click on your virtual machine. A new window with a graphical console of your VM will open.
2. The second button along the top 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 virtual machine's network card's MAC address. If you click on this, you can configure the virtual machine to use your new bridge.
4. Under the Network source drop-down menu, select the option Specify shared device name. This will allow you to type an interface name in the Bridge name text box; type `br0` into that text box. Make sure that the Device model is set to virtio.
5. Finally, click on Apply. You may have to restart the virtual machine for the changes to take affect:



Configuring a virtual machine to use bridge br0

While creating a brand new virtual machine, 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, 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:



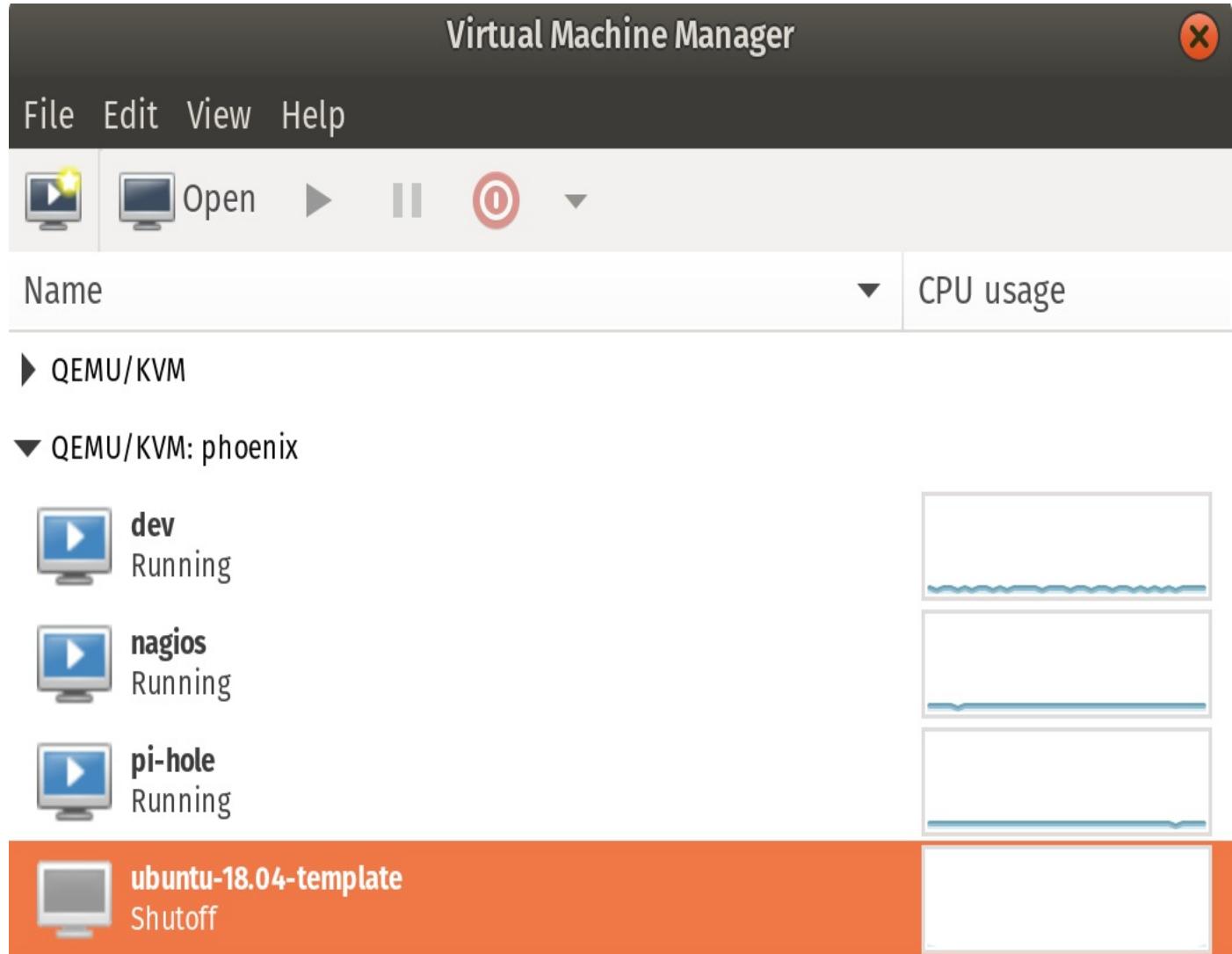
From this point onwards, you should have not only a fully-configured KVM server or instance, but also a solution that can be treated as a full citizen of your network. Your VMs will be able to receive an IP address from a DHCP server and communicate with other network nodes directly. If you have a very 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.

Simplifying virtual machine creation with cloning

Now that we have a KVM server, and we can spin up an army of virtual machines 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 again. We'll select an ISO file, navigate through the various screens to install the operating system, and then the VM is ready for use.

Most prominent virtualization solutions feature something called a **Template**. Essentially, we can create a virtual machine 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 many 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).

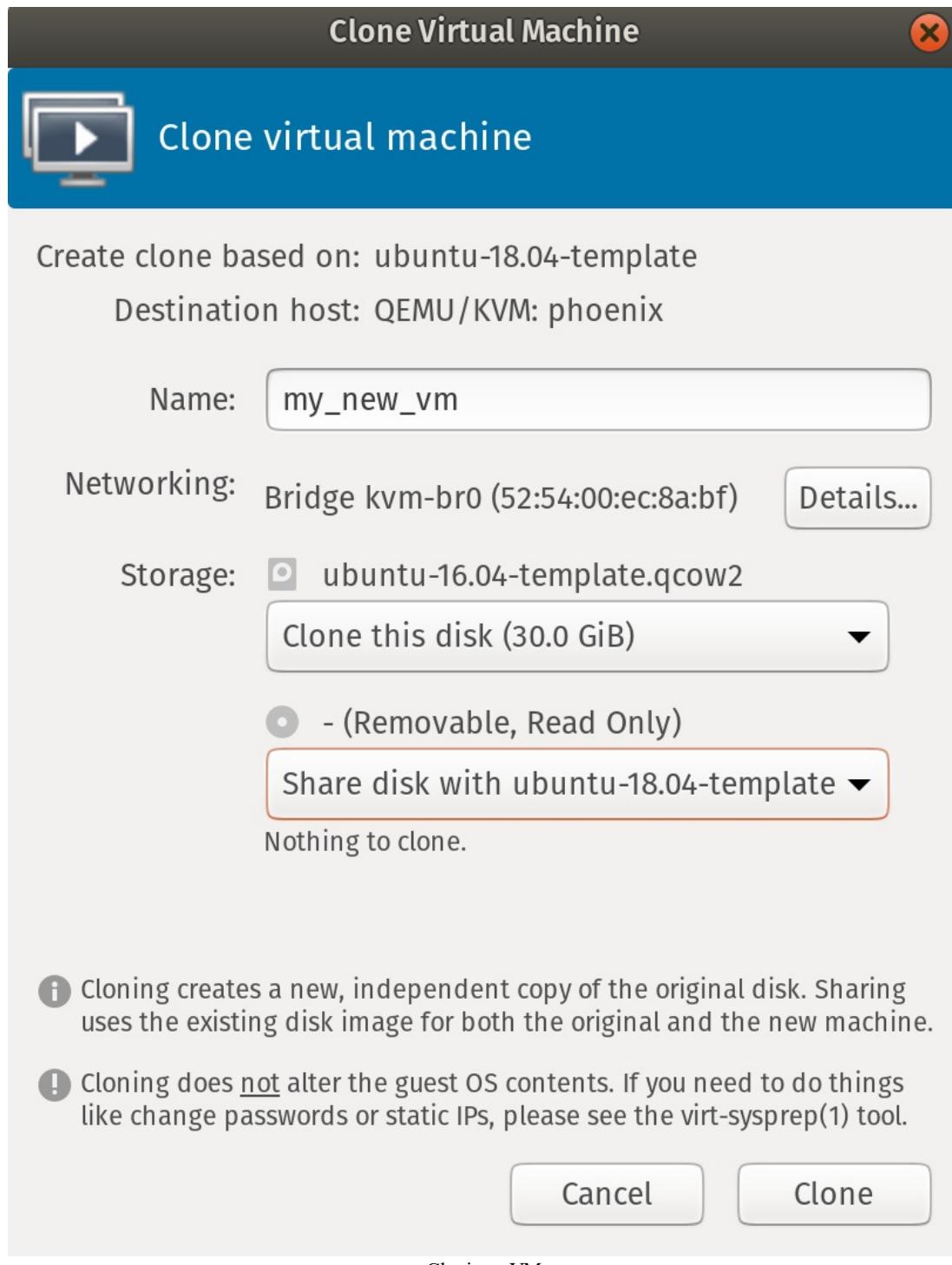
Unfortunately, as great as QEMU/KVM is, it doesn't have a template feature. This glaring hole in its feature set is a sizable setback, but thankfully we Linux administrators are very clever, and we can easily work around this to roll our own solution and create a system that is essentially the same thing as templates. Take the following screenshot, for example:



Selecting a bridge for a newly created VM

That screenshot is taken from an actual QEMU/KVM server that I maintain. It has four VMs, dev, nagios, pi-hole, and ubuntu-18.04-template. The latter is not a template at all;

it's just a virtual machine. There's nothing really different between it and the others, 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 virtual machine, I simply right-click on it, then click Clone. The following window will appear:



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 virtual machine. 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.

However, there are a few tasks to do after cloning the template. For one, it won't have all of the security updates that have been released since it was created, so updating all of the packages in the clone is a good first step so we can make sure it's current. But there's another aspect that you may not immediately think of, and that is OpenSSH. When installed, OpenSSH (assuming you had it installed on the original template) will generate a **Host Key**. On the clone, the OpenSSH host key will be the same as the one in the template. This isn't good, because that will confuse the OpenSSH client into thinking that it is connecting to the same machine but at a different IP address. Resetting the OpenSSH host key should be done before we put the new server into production. To do that, connect to the cloned machine and run the following two commands:

```
| sudo rm /etc/ssh/ssh_host_*
| sudo dpkg-reconfigure openssh-server
```

Basically, all you're doing is deleting all the original host keys and regenerating them. While this solution isn't glamorous, it does work. Now, the cloned virtual machine has its own set of host keys. Now you can put this server into production. So long as you maintain your base VM, you can spin up as many virtual machines as you need and be able to do so with minimal configuration steps.

Managing virtual machines via the command line

In this chapter, I showed you how to manage virtual machines with `virt-manager`. This is great if you have a secondary machine with a graphical user interface 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 virtual machine or checking to see which virtual machines are running on the server?

On the virtual machine server itself, you have access to the `virsh` suite of commands, which will allow you to manage virtual machines even if a GUI isn't available. To use these commands, simply connect to the machine that stores your virtual machines via SSH. What follows are some easy examples to get you started. Here's the first one:



```
| virsh list
Terminal
12:31:13 [0] [phoenix:~] % virsh list
Id      Name                State
-----
1       nagios              running
2       dev                 running
3       pi-hole             running
12:31:38 [0] [phoenix:~] %
```

Showing running virtual machines with the `virsh list` command

With one command, we were able to list which virtual machines are running on the server. In the example screenshot, which is taken from my very own KVM server, you can see that I have three virtual machines running there. 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 virtual machines with any of the following commands:

```
| virsh start my_vm
| virsh shutdown my_vm
```

```
| virsh suspend my_vm  
| virsh resume my_vm  
| virsh destroy my_vm
```

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 virtual machine. The last command is very destructive, as we'd use it when we want to permanently delete a VM from our server. You should be **VERY** careful with that command.

That's not all `virsh` can do, however. We can actually create a virtual machine 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 virtual machines can be accessible from the rest of the network and created our first virtual machine. In addition, although QEMU/KVM doesn't have its own solution for templating, we worked around that and created our own solution.

In [Chapter 13](#), Running Containers, we'll take a look at containerization, which will include both Docker and LXD. Stay tuned!

Questions

1. Between KVM and QEMU, which one handles hardware emulation, and which handles the Linux kernel instructions?
2. Which application can be installed in order to interface with your QEMU/KVM server?
3. A _____ is a directory on the filesystem that can store either virtual machine disk images, as well as ISO images for installing the operating system.
4. A _____ can be created to allow your virtual machines to be accessible to the rest of your network.
5. What are some tasks you should perform after cloning a virtual machine?
6. What are some of the requirements in order to be able to run virtual machines on your hardware?

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>

Running Containers

The IT industry never ceases to amaze me. First, the concept of virtualization came about and revolutionized the data center. Virtualization allowed us to run many smaller virtual machines 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 it also changed the way we develop them. In this chapter, we will cover the exciting world of containerization. This exploration will include:

- What containerization is
- Understanding the differences between Docker and LXD
- Installing Docker
- Managing Docker containers
- Automating Docker image creation with Dockerfiles
- Managing LXD containers

What is containerization?

In the last chapter, we covered virtualization. Virtualization allows us to run multiple virtual servers in 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 virtual machine 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 virtual machine. 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 as well. Nowadays, virtual machine solutions do have ways of sharing unused resources, but effectively, resource efficiency is a natural weakness of the platform.

Containers, unlike virtual machines, are not actual servers. At least, not in the way you typically think about them. While virtual machines 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 like a virtual machine 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 it all in a single segregated directory, and having the binary contents of the filesystem executed as a program, without an actual operating system running. This would mean that a container would use only the resources it needed.

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 virtual machines 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 a huge technology and has many benefits. It's definitely worth using, and you may even find yourself preferring it to virtual machines.

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. Virtual machines 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 a web application (Jenkins, and so on) 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.

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. You can't visit a single IT conference nowadays without at least hearing about it. 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.

LXD (pronounced **Lex-D**) finds its roots in LXC, so it's important to understand that first before we talk about LXD. **LXC** is short for **Linux Containers** (pronounced **Lex-C**) 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, at least, in part, specific to Ubuntu. That's not completely true these days, as the technology has been made to work on other platforms, too. This is due to the fact that the LXD software is distributed via `snap` packages, so any distribution of Linux that is able to install `snap` packages should be able to install LXD. Created initially by Canonical (the company behind Ubuntu itself) LXD enhances 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 (you even utilize the `lxc` command to manage it). Perhaps the best way to think of LXD is 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 virtual machine while Docker tries harder to differentiate itself from that. By 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. Essentially, LXC 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 virtual machine, and a Docker as an application container that provides the foundation needed to run an application. Regardless of these differences, both technologies can be used 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 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 heavy 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)**.

Another benefit of Docker is the **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. However, always make sure to audit third-party resources before you put them to use in your organization.

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 KVM virtualization server up and running as well as tweaking some configuration files. By comparison, installing Docker is effortless, as you only need to install the `docker.io` package:

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

Docker is only supported on 64-bit versions of Ubuntu. The required package is available in 32-bit Ubuntu, but it's not guaranteed to function properly. If you don't recall which version of Ubuntu Server you installed, run the following command:



You should receive the following output:

```
x86_64
```

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 running on your machine, simply titled `docker`. You can inspect it with the `systemctl` command, as with any other service:

```
| systemctl status docker
```

You should see that the service is running, and enabled to start at boot. 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, your user account is a member of the `docker` group, so

you're good to go. Wow, that was easy!

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 the 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 virtual machine 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 the Docker Hub, or you can choose to be selfish and keep it to yourself. 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 the 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:

```
Terminal
22:33:32 [0] [seraph:~] % docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
1be7f2b886e8: Pull complete
6fbc4a21b806: Pull complete
c71a6f8e1378: Pull complete
4be3072e5a37: Pull complete
06c6d2f59700: Pull complete
Digest: sha256:e27e9d7f7f28d67aa9e2d7540bdc2b33254b452ee8e60f388875e5b7d9b2b696
Status: Downloaded newer image for ubuntu:latest
22:33:57 [0] [seraph:~] %
```

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. The output will look similar to this:

```
| docker images
Terminal
22:33:57 [0] [seraph:~] % docker images
REPOSITORY      TAG          IMAGE ID   CREATED        SIZE
ubuntu          latest       0458a4468cbc   3 weeks ago   112 MB
22:36:19 [0] [seraph:~] %
```

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 0458a4468cbc
 Feel free to remove the Ubuntu image if you 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. The application you're running 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:latest /bin/bash
```

Once you run that command, you'll notice that your shell prompt immediately changes. You're now within 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 exit the container.

Every container you run has a container ID that differentiates it from others. If you want to remove a 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. Exiting the container this way, though, will remove it. 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 a 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. 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
```

Normally, we use the `-it` options to create a container. This is what we used a few pages back. 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
```

Now we have Apache installed. 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 `Ctrl + Q` 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 used an `ENTRYPOINT` of `/bin/bash`. While that's perfectly valid, `ENTRYPOINTS` are generally Bash scripts that are configured to run the desired application and are 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. Before we do so, we should configure Apache to automatically start when the container is started. We'll do this a bit differently inside the container than we would on an actual Ubuntu Server. Attach to the container and open the `/etc/bash.bashrc` file in a text editor within the container. In order to do this, you may need to install a text editor (such as `nano`) with the `apt` command, as the container may not have an editor installed:

```
| sudo apt install nano
```

Add the following to the very end of the `/etc/bash/rc` file inside the container:

```
| /etc/init.d/apache2 start
```

Save the file and exit your editor. Exit the container with the Ctrl + P and Ctrl + Q key combinations. Next, let's grab the container ID 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 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. This will send the `SIGTERM` signal to the container, followed by `SIGKILL` if it doesn't stop on its own after a delay:

```
| docker stop <Container ID>
```

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 GCC 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 baked in that they needed to perform their job. 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 OS was, they all had the same tools. This gave each developer the freedom to run their preferred distribution of Linux (and the stranger ones used macOS) and it didn't impact their ability 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>

# Update the container's packages
RUN apt update; apt dist-upgrade -y

# Install apache2 and vim
RUN apt install -y apache2 vim

# Make Apache automatically start-up
RUN echo "/etc/init.d/apache2 start" >> /etc/bash.bashrc
```

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 the 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.

```
| # Update the container's packages
```

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.

```
| 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 the command runs.

```
| 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.

```
| RUN echo "/etc/init.d/apache2 start" >> /etc/bash.bashrc
```

In the previous section, we copied the startup command for the `apache2` daemon into the `/etc/bash.bashrc` file to serve as an example of how to automatically start an application. We're including that here so we won't have to do this ourselves when containers are created from the image. Naturally, you'd want to write an `ENTRYPOINT` script, but that's a relatively advanced topic and we don't really need to focus on that to get an understanding of how Docker works.

Great, so 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 /bin/bash
```

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:~$ lxd init
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 (dir, btrfs, lvm) [default=btrfs]:
Create a new BTRFS pool (yes/no) [default=yes]?
Would you like to use an existing block device (yes/no) [default=no]?
Size in GB of the new loop device (1GB minimum) [default=15GB]: 2
Would you like LXD to be available over the network (yes/no) [default=no]?
Would you like stale cached images to be updated automatically (yes/no) [default=yes]?
Would you like to create a new 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]?
LXD has been successfully configured.
```

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 fine for everything, and for the size of the pool, I just used 2 GB

but you can use whatever size you want to. Even though we chose the defaults for each 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:18.04 mycontainer
```

With that simple command, LXD will now download the root filesystem for this container and set it up for us. Once done, the container will actually be 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` and based it on Ubuntu 18.04. The process should be fairly easy to follow so far.

You might be wondering why we used an `lxc` command to create a container, since we're learning about LXD here. As I mentioned earlier, LXD is a management layer on top of 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	<code>lxc list</code>

Start a container	<code>lxc start <container></code>
Stop a container	<code>lxc stop <container></code>
Remove a container	<code>lxc delete <container></code>
List downloaded images	<code>lxc image list</code>
Remove an image	<code>lxc image delete <image_name></code>

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
```

The preceding command immediately logs you in to 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 out when you exit a container, and you don't have to exit it a certain 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 -- sudo --login --user 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 to 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
```

Now your newly created container will 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. In case 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 virtual machines, 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 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 [Chapter 12](#), Virtualization. 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 12](#), 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 extbr0 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 network 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:18.04 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
```

At 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 virtual machines. 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 venture back into the world of automation. While we have taken a look at scripting in [chapter 11](#), Learning Advanced Shell Techniques, we're going to look at an even more powerful method in [Chapter 14](#), Automating Server Configuration with Ansible. There, we'll venture into the exciting world of Ansible.

Questions

1. Name two of the solutions you can use to run containers.
2. What are some of the pros and cons to containerization?
3. What is the name of the online registry where Docker containers come from (assuming you are using only defaults)?
4. LXD is an extension of which containerization technology?
5. What company is behind LXD and is the steward of its development?
6. A _____ allows you to automate the building of a Docker container.
7. Name some of the extra features that LXD provides that aren't normally part of LXC.

Further reading

- **Docker documentation page:** <https://docs.docker.com/>
- **Canonical's LXD web page:** <https://www.ubuntu.com/containers/lxd>
- **LXD documentation:** <https://help.ubuntu.com/lts/serverguide/lxd.html>

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 minimal of 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 a web server deployment
- Using Ansible's pull method

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 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**, 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.

Infrastructure as Code, 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 11](#), Learning Advanced Shell Techniques, 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 feature 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?

Most 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 master 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 master and client.

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 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. 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.

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 the files for your project, including the revision history.

Owner	Repository name
 jlacroix82 	/ ansible 
Great repository names are short and memorable. Need inspiration? How about ubiquitous-broccoli .	
Description (optional)	
Some awesome Ansible playbooks	
<input checked="" type="checkbox"/> Public Anyone can see this repository. You choose who can commit.	
<input type="checkbox"/> Private You choose who can see and commit to this repository.	
Initialize this repository with a README	
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.	
Add .gitignore: None 	Add a license: None  
Create repository	

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, but there is a charge for that. Since we're not going to include confidential information in the repository, we won't need to worry about that.

Once the repository is created, 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 administrators use to provision servers will be placed inside a repository for safe-keeping. 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 often offer an older version of Ansible in their repositories, with Ubuntu being no exception. This 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. I think it's better to get the software from the developers themselves.

The examples in this book were created with Ansible 2.5 in mind. However, depending on when you are reading this, a new version is most likely available. This shouldn't be an issue in regards to the examples in this chapter, because Ansible handles backwards 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).

So, what do we do? Simple: we add a PPA to our server that is maintained by the developers of Ansible themselves. This repository will always give you the latest stable version that's available. Let's go ahead and get this wonderful tool installed. We'll install Ansible on a central server (or even just our workstation). We won't need to install Ansible on the individual hosts that we'll maintain. As I mentioned, there's no agent with Ansible, so there's nothing to install on the clients.

First, we will add this PPA to our central server or workstation:

```
| sudo apt-add-repository ppa:ansible/ansible
```

Next, let's update our package indexes:

```
| sudo apt update
```

Then, we'll install Ansible itself:

```
| 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 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 4](#), Connecting to Networks, so refer back to that if you need a reminder. Creating users was covered in [chapter 2](#), Managing Users. 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 server or local machine.
4. Set up the Ansible user on the server such that it can connect to clients via SSH without a password.
5. Configure `sudo` on the client machines such 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. 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.

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.

With the inventory file, I highly recommend that you don't include it in your Ansible Git repository. Reason being, if a miscreant is looking at it, they will be able to glean important information regarding the layout of your internal servers. This file is only needed on the server that's controlling the others anyway, so it shouldn't be an inconvenience to not have it in the repository.

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, populate the file with the IP addresses of the servers you wish to manage. 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 those 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.

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/ansible.cfg`, 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 that 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 of 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:

```
192.168.1.145 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Depending on how many servers you have set up, you should see that output one or more times. 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. Now that the communication exists, we can actually build some actual configuration.

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 itself 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
```

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 were 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 one. Then, you can have configuration only be applied to hosts inside a particular role. We're not going to get into that in this chapter, but just know that it is possible:

```
| 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:

```
| tasks:
```

This line starts the next section, which is where we place our individual tasks:

```
| - 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:

```
| apt: name=htop
```

Next, we utilize the `apt` module and tell it to install a package, `htop` in this case. 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 more 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, I think that is very sloppy. 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={{item}}
      with_items:
        - htop
        - git
        - 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 as well. 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
```

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 a package, but also start the service for it. You can also do things such as keep packages up to date, copy a configuration file, set up a template, and so 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 have the `apache2` service started:

```
---
```

```
- 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` service. (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 will take care of the specifics for you.

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 of 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:

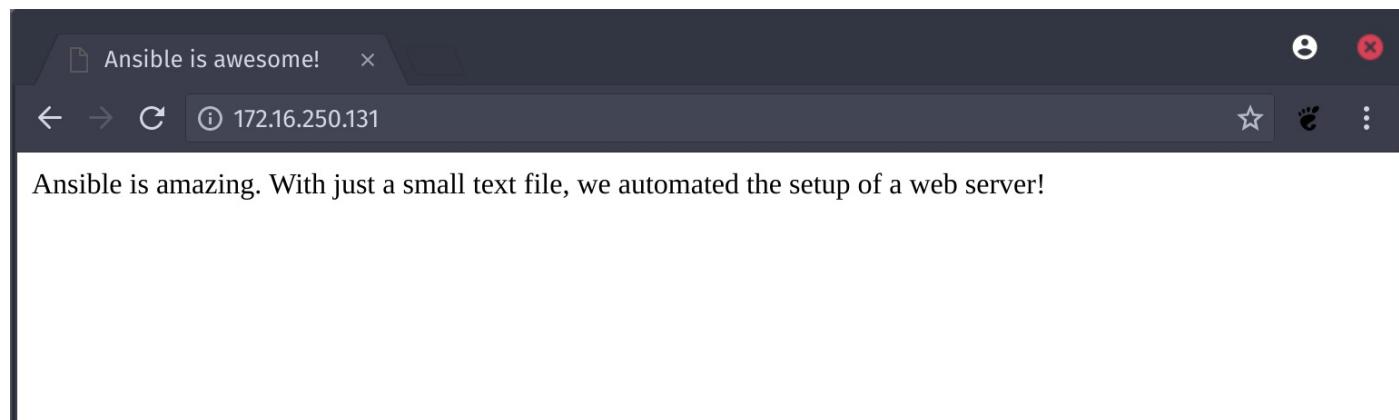
```
---
- hosts: all
  become: true
  tasks:
    - name: Install Apache
      apt: name=apache2
    - name: Start the apache2 services
      service: name=apache2 state=started
    - name: Copy index.html
      copy: src=index.html dest=/var/www/html/index.html
```

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 a source (`src`) and destination (`dest`).

Let's go ahead and run the new Playbook:

```
| ansible-playbook apache.yml
```

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:



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 will include guides on how to do all sorts of things and utilize modules 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. This won't be an issue from a resource perspective, since Ansible won't be actually 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 during 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 we wish for it to manage. To add a new server, we create the user account and 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 desired to do so. However, that's out of the scope of this book, and there's an easier method anyway.

As you know, normally 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 themselves. 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 actual 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 below:

```
---  
- 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 h 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 their 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. Full coverage of AWS is beyond the scope of this book, but it's important 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 is already 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 downloading the repository locally. 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, 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 the most important one yet. We'll take a look at security, which is something you'll want to pay special attention to, the reason being, security makes or breaks our organization, and we want to make sure we do everything in our power to keep the bad guys out of our environment. During our journey into security, we'll secure SSH, take a look at encryption, lock down `sudo`, and more. See you there!

Questions

1. Other than Ansible, what are some examples of other configuration management utilities?
2. What are some advantages Ansible has over its competition?
3. Ansible reads a list of hosts from its _____ file.
4. Ansible utilizes _____ to communicate with hosts.
5. A _____ is a collection of individual tasks, known as _____, which Ansible performs against its targets.
6. Name some of Ansible's modules, which we've used in this chapter.
7. By looking through Ansible's documentation, list some additional modules that can be used, as well as what they allow you to do.
8. Describe the difference between the inventory and pull methods of Ansible.

Further reading

- **Ansible documentation:** <http://docs.ansible.com/ansible/latest/index.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/Getting-Started-Git-Basics>
- **Set up Git (GitHub):** <https://help.github.com/articles/set-up-git/>

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 every day, but someone has to install them. OpenSSH is a very handy utility, but it's also the first point of entry for an attacker who finds an insecure installation. Backups are a must-have, but are also the bane of a company that doesn't secure them and then the data can 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`

Lowering your attack surface

After setting up a new server, 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 can 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 `netstat` command:

```
| sudo netstat -tulpn
jay@ubuntu:~$ sudo netstat -tulpn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State      PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN     693/sshd
tcp        0      0 0.0.0.0:5355            0.0.0.0:*               LISTEN     446/systemd-resolve
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN     4832/rpcbind
tcp6       0      0 :::22                  :::*                  LISTEN     693/sshd
tcp6       0      0 :::5355                :::*                  LISTEN     446/systemd-resolve
tcp6       0      0 :::111                 :::*                  LISTEN     4832/rpcbind
tcp6       0      0 :::80                  :::*                  LISTEN     3628/apache2
udp        0      0 0.0.0.0:768             0.0.0.0:*               LISTEN     4832/rpcbind
udp       52224  0 127.0.0.53:53            0.0.0.0:*               LISTEN     446/systemd-resolve
udp        0      0 172.16.250.130:68          0.0.0.0:*               LISTEN     443/systemd-network
udp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN     4832/rpcbind
udp       47616  0 0.0.0.0:5355            0.0.0.0:*               LISTEN     446/systemd-resolve
udp6       0      0 :::768                 :::*                  LISTEN     4832/rpcbind
udp6       0      0 :::111                 :::*                  LISTEN     4832/rpcbind
udp6      6144   0 :::5355                :::*                  LISTEN     446/systemd-resolve
jay@ubuntu:~$
```

Running a port query with netstat

You don't actually need to run the `netstat` command with `sudo` as I have, but if you do, the output will show more information than without running it, namely the names of the

programs that are listening for connections (which makes it easier to associate ports, if you don't have them all memorized). 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. The server I took the screenshot from has several of these open ports. 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 `netstat` 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 with some sort of security settings (which will depend on the particular service), or enabling a firewall, which we'll get to a bit later. 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.

This leads me to the very first step in lowering your attack surface: uninstalling unneeded packages. In my case, the server has several RPC services listening for connections. These services have to do with NFS, something that this server will never be serving or using in any way. Therefore, there's no harm in removing support for NFS from this server. It doesn't depend on it at all:

```
| sudo apt remove rpcbind
```

Now that I've taken care of RPC, I'll look at other services this server is running that are listening for connections. OpenSSH is a big one. It's usually going to be the first target any attacker uses to try to gain entry into your server. What's worse, this is usually a very important daemon for an administrator to have listening. After all, how can you administer the server if you can't connect to it? But, if you're able to connect to it, then conceivably so can someone else! What should we do? In this case, we'll want to lock the server down as much as possible. In fact, I'll be going over how to secure SSH later in this chapter. Of course, nothing we do will ever make a server bulletproof, but using the most secure settings we can is a great first step. Later on in this chapter, I'll also go over Fail2ban, 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 be the target of some sort of vulnerability that may be affected by some other port or means. Therefore, I'll say it again: remove any packages you don't need. 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 for 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. 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.

Another thing to keep in mind is using 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 absolutely must have a service listening for outside connections, 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.

Finally, it's important to employ the **principle of least privilege (PoLP)** 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 services and users, and allow them access to only perform the functions a user's job absolutely requires of them. This may involve, but is certainly not limited to:

- Adding a user to the fewest 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.

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, or Common Vulnerabilities and Exposures, is a special online catalog 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 18.04 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 out-of-date packages 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 as 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:

```
| sudo apt upgrade
```

```
jay@ubuntu: ~
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
The following packages will be upgraded:
  ethtool grub-legacy-ec2 hdparm isc-dhcp-client isc-dhcp-common lan
guage-pack-en publicsuffix
  python3-distupgrade ubuntu-release-upgrader-core
9 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Need to get 1,035 kB of archives.
After this operation, 311 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

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 9 packages. If you were to press Y and then 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.

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 regards 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 because 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 select 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 rollback 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-generic_4.15.0.7.8_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.

Automatically installing patches with the Canonical Livepatch service

Since the publication of the first edition of this book, Canonical has released a new 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 systems 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. You can, however, install the Livepatch service on **three** of your servers without paying, so it's still something you may want to consider. In my case, I simply have this applied to the three most critical servers under my jurisdiction, and the rest I update manually. 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 were catastrophic, 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. 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.

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 an additional cost penalty. At the time of writing, the current cost is \$0.01 USD per hour for the hosted version. 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 bottom-tier **Virtual Private Servers (VPS)** from a provider such as DigitalOcean will simply not cut it. At the minimum, you'll want 2 GB of RAM, and 1 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. Last, Landscape will only work 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 publication, Landscape server is not available on Ubuntu 18.04 yet, as Ubuntu 16.04 is the highest supported version. This is expected to be available at some point in the future, but for now, use

– Ubuntu 16.04 if you would like to utilize Landscape server

Setup of the Landscape server is relatively easy, there are only three commands to run. The following website 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 ppa:landscape/17.03  
sudo apt update  
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

Not secure | <https://ubuntu/new-standalone-user>

Welcome to Landscape

Name *

The first and last name.

E-mail address *

Personal e-mail address.

Passphrase *

A secret passphrase.

Verify passphrase *

Verify the secret passphrase.

Sign up

Setting up Landscape

Next, 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:

The screenshot shows the Landscape web interface. At the top, there's a header bar with tabs for 'Organisation', 'Computers', and 'OpenStack'. On the right side of the header, there's a notification icon (red triangle with '1') and user information for 'Malcolm Reynolds'.

In the main content area, there's a sidebar on the left with a 'computers registered' count of 0, remaining registrations of 60, and a link to register new computers. The main panel has a title 'Organisation' and displays a table with the following data:

Account name:	standalone
Registered computers:	0
Remaining full registrations:	10
Registered VMs:	0
Remaining VM registrations:	0
Registered containers:	0

To the right of the table, there are two sections: 'Activities waiting for approval' (empty) and 'Activities in progress' (empty). The URL in the browser address bar is <https://ubuntu/account/standalone>.

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/computers
```

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>/account/standalone
```

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.

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. It may be a good idea to benefit from that feature, but it's not required.

After you set up the client on your server, you should see a computer listed that is waiting for approval on the Landscape page. 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	Today 22:29 EST

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:

The screenshot shows the Landscape web interface. At the top, there's a header bar with the title '(2) - Select computers' and a URL 'https://ubuntu/account/standalone/computers'. The header also includes a user icon, a notification badge ('2'), and links for 'Malcolm Reynolds' and 'Logout'. Below the header is a navigation bar with tabs for 'Organisation', 'Computers' (which is selected), 'OpenStack', and other options. On the left, there's a sidebar with a search bar, a 'global (1)' section, and an 'Access Groups' section. The main content area shows a table titled 'Select computers' with one row. The table has columns for 'Name', 'Hostname', and 'Last ping time'. The single row shows 'My Web Server' with 'ubuntu' as the hostname and 'Today at 23:36 EST' as the last ping time. There's also a 'Download as CSV' button. At the bottom of the page is a footer with links for 'CANONICAL', 'Help', 'Latest News', 'API Documentation', and 'User Guide'.

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:

Info Activities Hardware Monitoring Scripts Processes Packages Users Reports

The Landscape computer menu

Here's a list of the common items in this menu and a short description of what each one does:

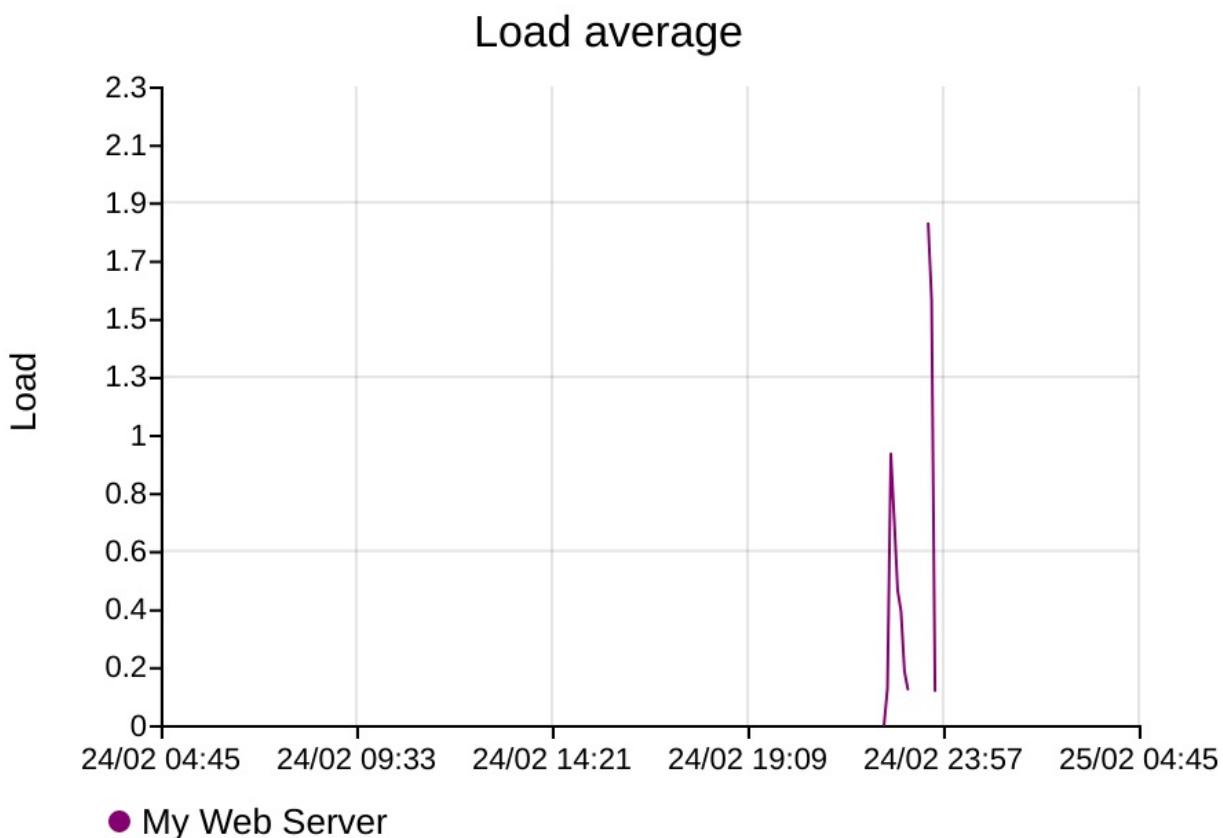
- **Info:** Shows basic information about the server, such as what model of processor it has, RAM, and other system information.
- **Activities:** The Activities section shows the status of any commands you've given the server to perform through the interface, such as installing a package. The following is an example screenshot of what the Activities page looks like when you've instructed a server to install a package:

Status	Description	Computers	Creator	Created at
	Install package vim-nox	My Web Server	Malcolm Reynolds	Today 23:43 EST

Example of an activity in Landscape

- **Hardware:** Provides more specific details about hardware devices installed on the server.
- **Monitoring:** This section allows you to view information regarding resource utilization:

Load



A resource graph in Landscape

- **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 also include a link to the official documentation pages at the end of this chapter if you would like to explore this service even more.

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 4](#), 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, that 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. 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.

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. 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. 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:

```
| AllowGroups admins sshusers gremlins
```

I recommend you use only one or the other. 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, 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 DigitalOcean, 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 4](#), 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 4](#), 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
```

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. In the past, I used a utility known as **DenyHosts** to secure OpenSSH. DenyHosts protects SSH (no more, no less). It watches the server's log files, looking for authentication attempts. If it sees too many failures from a single IP address, it will create a firewall rule to block that IP. The problem was that it only protected OpenSSH. Another problem is that DenyHosts just kind of went away quietly. For some reason, it stopped being maintained and some distributions removed it outright. Fail2ban does what DenyHosts used to do (protect SSH) and more, as it is able to protect other services as well.

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 54 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 63 includes the `bantime` option. This option pertains to how many seconds a host is banned when Fail2ban blocks it. This option defaults to `10m`:

```
| 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, until 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 231 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, 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 the 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 /etc/fail2ban/jail.local file:

```
| enabled = true
```



Earlier versions of Ubuntu included the `enabled = false` line for each Jail in the sample `jail.conf` file. In that case, you'd only need to change the line to `enabled = true` to enable a Jail if you're using an older version of Ubuntu.

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 exit, 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, enable `[apache-shellshock]` while you're at it. 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 whitelist your own networks, in case you accidentally

type your own SSH password incorrectly too many times. 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.

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, 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, 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 do the same but 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:

```
| ALL: 192.168.1.50
```

This rule allows a machine with an IP address of `192.168.1.50` to access the server:

```
| ALL: 192.168.1.0/255.255.255.0
```

This rule allows any machine within the `192.168.1.0/24` network to access the server:

```
| ALL: 192.168.1.
```

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: ALL
```

This rule allows everything. You definitely don't want to do this:

```
| ssh: 192.168.1.
```

We can also allow specific daemons. Here, I'm allowing OpenSSH traffic originating from any IP address beginning with `192.168.1`.

On your end, if you wish to utilize this security approach, add the resources on your 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 setup, 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 9](#), 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 mysampled.* TO 'appuser'@'localhost' IDENTIFIED BY 'password';
```

What's important here is that we're allowing access to the `mysampled` 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 mysampled.* 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 mysampled.* 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 mysampled.* 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 on Pastebin (<https://pastebin.com/>). A good security design includes regular patching.

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/bionic/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 executed so far in this section trigger the `iptables` command, which is a command 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
```

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 attackers would have to bypass in order to do harm.

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 **Linux Unified Key Setup (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 also 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. 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
```

If we wish to change the passphrase, we can use the following command. Keep in mind that you should absolutely be careful typing in the new passphrase, so you don't lock yourself out of the drive. 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.

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.

Locking down sudo

We've been using the `sudo` command throughout the book. In fact, we took a deeper look at it during [Chapter 2](#), Managing Users. 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, 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
```



This line is configured via the `visudo` command, which we covered in [Chapter 2](#), Managing Users.

For the most part, if a user needs access to `sudo`, just give them access to 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, `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 can 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 **Uncomplicated Firewall (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.

Questions

- What is a CVE, and what is its purpose?
- Applying updates to a running kernel can be done automatically and without rebooting by utilizing Canonical's _____ service.
- The default configuration file for Fail2Ban is _____, but you should copy it to _____ to avoid your changes being overwritten.
- What are some of the things you can do to secure OpenSSH?
- Explain the principle of least privilege.
- What command can we use in order to list the ports our server is listening on?
- The _____ command allows us to encrypt an entire disk.
- Explore Landscape on a test server, and list some of the features you're able to find within the interface that weren't mentioned in this chapter.

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>

Troubleshooting Ubuntu Servers

So far, have we covered many topics, ranging from installing software to setting up services to provide value to our network. But what about when things go wrong? 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 something goes wrong and a process isn't behaving quite as you expect it to. In this chapter, we'll take a look at some common starting points and techniques when it comes to troubleshooting issues with our servers.

In this chapter, we will cover:

- Evaluating the problem space
- Conducting a root cause analysis
- Viewing system logs
- Tracing network issues
- Troubleshooting resource issues
- Diagnosing defective RAM

Evaluating the problem space

After you identify the symptoms of the issue, the first goal in troubleshooting is to identify the problem space. 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 problem space 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 in regard to its ability (or inability) to do the job designated for it. In other cases, the problem space 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. This will also help you identify the scope.

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, since each of your users will be under the impression that their issues are more important than anyone else.

When identifying the problem space, as well as 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 around the network around that same time?

- Has this problem happened before? If so, what was done to fix it the 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 checking 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.

Conducting a root cause analysis

Once you solve 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 in into the server and check the logs, you can see that `Bob` not only logged into the server near the time of the incident, his bash history literally shows him running the `rm -rf /work/important-files` command. At this point, 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 servers that were "fencing." At a company I once worked for, our Citrix-based virtual machine 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 as 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 example 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, I also wasn't the individual who set them up, whom had already left the company. There was no way I could reach a 100 percent 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 the question "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 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 virtual machine 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 for the reason 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.

Viewing system logs

After you identify the problem space, you can attack the potential origin of the problem. Often, this will involve reviewing log files on your server. Linux has great logging, 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, which may give you a hint as to what may be going on if the problem is related to your web server. MySQL and MariaDB create their log files in the `/var/log/mysql` directory. These are known as **Application Logs**, which are basically log files created by an application and not the distribution. There are also **System Logs**, such as the authorization or system logs. System logs 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 following commands 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 happens.

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 their 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 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 same 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 the `/var/log/auth.log` mentioning such if that were the case.

Another use case for checking the `/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 brute-force logging in to the server. 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 time-stamp 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 the `auth.log` and the `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.

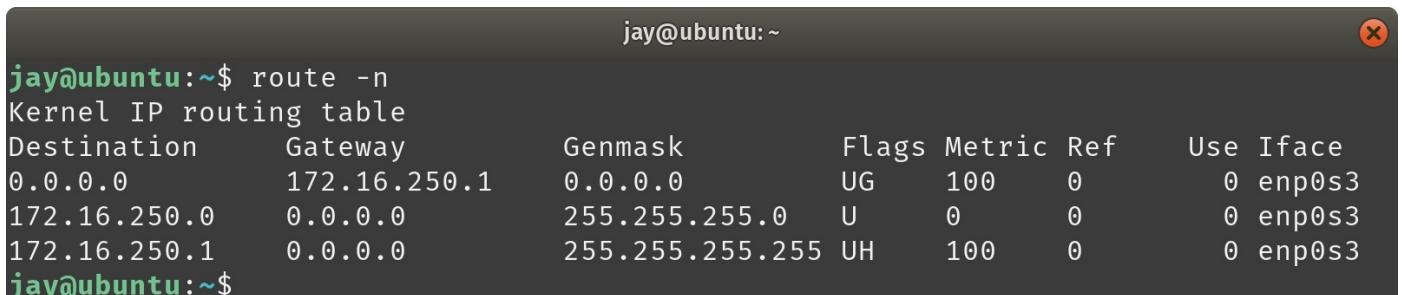
As you can see, Ubuntu contains very helpful log files which will aid you in troubleshooting your servers. Often, when you're faced with a problem, viewing relevant log entries and then conducting a Google search regarding them will result in a useful answer, or at least bring you to a bug report to 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.

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 `route -n` command. This command will print your current routing table information:



```
jay@ubuntu:~$ route -n
Kernel IP routing table
Destination      Gateway         Genmask        Flags Metric Ref    Use Iface
0.0.0.0          172.16.250.1   0.0.0.0        UG    100    0        0 enp0s3
172.16.250.0     0.0.0.0        255.255.255.0  U     0    0        0 enp0s3
172.16.250.1     0.0.0.0        255.255.255.255 UH    100    0        0 enp0s3
jay@ubuntu:~$
```

Viewing the routing table on an Ubuntu Server

In this example, you can see that the default gateway for all traffic is `172.16.250.1`. 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 `172.16.250.1`. 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. 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:

```
| 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 a 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 7](#), 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 7](#), 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. The `dig` command is also useful for troubleshooting caching. The first time you use the `dig` command, you'll see a response time (in milliseconds). The subsequent time you run it, the response time should be much shorter:

```
jay@ubuntu:~$ dig packtpub.com

; <>> DiG 9.11.2-P1-1ubuntu3-Ubuntu <>> packtpub.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34930
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;packtpub.com.           IN      A

;; ANSWER SECTION:
packtpub.com.        7195    IN      A      83.166.169.231

;; Query time: 0 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
;; WHEN: Sun Feb 25 08:38:08 EST 2018
;; MSG SIZE  rcvd: 57

jay@ubuntu:~$ host packtpub.com
packtpub.com has address 83.166.169.231
packtpub.com mail is handled by 5 imap.packtpub.com.
packtpub.com mail is handled by 10 mx-capricab.easydns.com.
jay@ubuntu:~$
```

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 18.04 LTS (which was released in April of 2018), it's able to

support hardware released as of the beginning of 2018 and older. Future releases of Ubuntu Server will publish hardware entitlement updates, which will allow Ubuntu Server 18.04 to support newer hardware and chip-sets once it comes out. Therefore, it's always important to use the latest installation media when rolling out a new server. Typically, Ubuntu will release several point releases during the life of a supported distribution, such as 18.04.1, 18.04.2, and so on. As long as you're using the latest one, you'll have 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  
          Network Controller  
  
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 which 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 relating 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 7, 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.

Troubleshooting resource issues

I don't know about others, but it seems that a majority of my time troubleshooting servers usually comes down to 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 3](#), 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 used and available inodes. The reason you should also run this, is because 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 3](#), 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 ncdū -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 that 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 issues 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 otherwise, 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 off-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:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:       481          91         62          0        326        373
Swap:      947           7         939
jay@ubuntu:~$
```

Output of the dig and host commands

In [Chapter 6](#), Controlling and Monitoring Processes, 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 `iostop`. While not installed by default, the `iostop` utility is definitely a must-have, so I recommend you install the `iostop` package. The `iostop` utility itself needs to be run as `root` or with `sudo`:

```
| sudo iostop
```

The `iostop` command will allow you to see how much data is being written to or read from your disks. Input/output 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 `iostop` 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:

jay@ubuntu:~							
Total DISK READ :			0.00 B/s		Total DISK WRITE :	0.00 B/s	
Actual DISK READ:			0.00 B/s		Actual DISK WRITE:	0.00 B/s	
TID	PRIOS	USER	DISK READ	DISK WRITE	SWAPIN	IO>	COMMAND
1	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	init
2	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[kthreadd]
4	be/0	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[kworker/0:0H]
6	be/0	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[mm_percpu_wq]
7	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[ksoftirqd/0]
8	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcu_sched]
9	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[rcu_bh]
10	rt/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[migration/0]
11	rt/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[watchdog/0]
12	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[cpuhp/0]
13	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	[kdevtmpfs]

The `iostop` utility running on an Ubuntu Server

The `iostop` 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, DISKWRITE, DISK READ, and others. When you're finished with the application, press Q to quit.

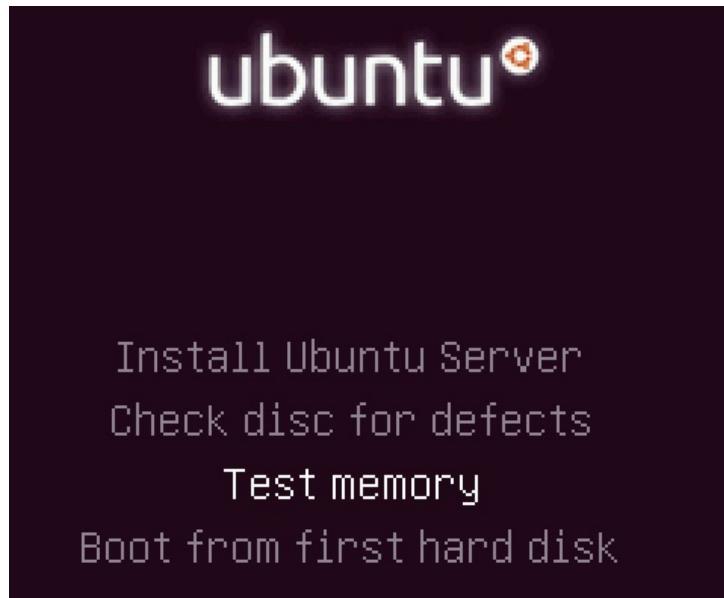
The utilities we looked at in this section are very useful when identifying issues with bottle-necked resources. What you do to correct the situation after you find the culprit will depend on the daemon. Perhaps there's an invalid configuration, or the daemon has encountered a fault and needs to be restarted. Often, checking the logs may lead you to an answer as to why a daemon misbehaves. In the case of a full storage, almost nothing beats `ncdu`, which will almost always lead you directly to the problem. Tools such as `htop` and `iostop` 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.

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 it's one of those situations where I can't give you a definite list of symptoms to look out for that will point to memory being the source of an issue you may be experiencing. 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 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 an option to Test memory:



The main menu of the Ubuntu installer, showing a memory test 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:

```
Memtest86+ 5.01 | Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
CLK: 2712 MHz (X64 Mode) | Pass 43% #####
L1 Cache: 32K 271212 MB/s | Test 73% #####
L2 Cache: 256K 90404 MB/s | Test #8 [Moving inversions, 32 bit pattern]
L3 Cache: 3072K 50224 MB/s | Testing: 1024K - 512M 511M of 512M
Memory : 512M 18965 MB/s | Pattern: 00040000 | Time: 0:00:31
-----
Core#: 0 (SMP: Disabled) | RAM: 1616MHz (DDR3-3233) - BCLK: 104
State: | Running... | Timings: CAS 19-15-15-31 @ 192-bit Mode
Cores: 1 Active / 1 Total (Run: All) | Pass: 0 Errors: 0
-----
```

Memtest86+ in action

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 assure 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 because 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 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 when problems occur and that you 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 problem space, 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 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 advanced 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.

Questions

1. Name some of the benefits of Bash history and how it can benefit the troubleshooting process.
2. A root cause analysis assists in the troubleshooting process in what way?
3. Name some of the important things to include in a root cause analysis.
4. System and application logs are stored in which directory?
5. Name some of the commands you can use in order to browse the contents of log files.
6. What are at least two commands you can use to troubleshoot DNS issues?
7. Which command can you use to list hardware devices stored on a server?

Preventing and Recovering from 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

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 can 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 regards 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 15](#), 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 around your company, and the types of activities each are 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 (as well as [Chapter 15](#), 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 it, 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 quickly 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 15](#), Securing Your Server, several times. In [Chapter 15](#), Securing Your Server, we looked into securing our servers. 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.

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, DHCP server configuration, DNS zone files, and more. If a server were to encounter a disaster from which the only recourse is to completely rebuild it, the last thing we'll want to do is re-engineer all of this configuration from scratch. This is where Git comes in.

Git is a development tool, used by software engineers everywhere for the purpose of version control for source code. 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 changes to configuration, we'll push the change 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 back to a known working commit and we'll be immediately back up and running.

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 14](#), 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 on it 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 on 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 2016-06-25 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 because 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` backwards) 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 repository for a Git repository on one of my servers:

```
2018-03-14 11:46 Jay LaCroix o [master] {origin/master} {origin/HEAD} fixed ip address for jenkins.  
2018-03-14 10:41 Jay LaCroix o fixed syntax for jenkins check.  
2018-03-13 13:59 Jay LaCroix o changed service description for pi-hole.  
2018-03-13 13:51 Jay LaCroix o added port check for jenkins/8080  
2018-03-13 08:38 Jay LaCroix o fixed hostgroup  
2018-03-13 08:36 Jay LaCroix o renamed vm to lxd  
2018-03-12 22:06 Jay LaCroix o updated for lxd.  
2018-03-12 22:05 Jay LaCroix o updated for lxd.  
2018-03-12 22:05 Jay LaCroix o initial commit  
2018-03-12 22:05 Jay LaCroix o renamed vm group to lxd.  
2018-03-09 15:52 Jay LaCroix o decommissioned host.
```

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 some syntax issues on 3/14/2018. If I want to restore that file, I should revert to the commit before that, on 3/13/2018. 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 master 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 will 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.

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'll 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 slave 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 very 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 as well. If we have another server to send the backup to, even better.

At one company I've 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 bit 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 has 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 also the experience was a good example of utilizing what you have available to you.

Since we've already gone over `rsync` in [Chapter 8](#), 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 8](#), 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 into 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 8/16/2018 would have a backup directory of the following path, if we used the command I gave as an example:

```
/backup/incremental/8-16-2018
```

This essentially allows you to keep differentials. Files on `/src` will still be copied to `/target`, but the directory you identify as a `--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 to. 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.

Replacing failed RAID disks

RAID is a very useful technology, as it can help your server survive through 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 the other 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
                jay@ubuntu:~
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:~$
```

A healthy RAID array

In this screenshot, 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 Us 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>
```

RAID status output with a faulty drive

As you can see from the screenshot, 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 can make this problem much worse than it already is. We can use the `hdparm` command to get a little more info from our drive. The following command will give us info regarding `/dev/sda`, the disk that's currently still functioning properly:

```
| sudo hdparm -i /dev/sda
/dev/sda:

Model=TOSHIBA DT01ACA200, FwRev=MX40ABB0, 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
```

Output of the `hdparm` command

The reason why we're executing this command against a working drive is because 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 label 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
<code>/dev/sda1</code>		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

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 /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 RAID status:

```
| cat /proc/mdstat
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
```

Checking RAID status after replacing a disk

In the output of the previous screenshot, 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 now rest easy.

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'll 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.

In addition, you can actually restore a system's ability to boot, using a third-party utility known as **Boot Repair**. This utility will allow you to restore GRUB on a system that has a damaged boot sector and is no longer able to boot. GRUB is a crucial service, since it's the boot-loader that's responsible for a system's ability to boot a Linux

distribution. Should GRUB ever stop functioning properly, the Boot Repair utility is a good way to fix it. Having to reinstall or repair GRUB is actually quite common on systems that dual boot some flavor of Linux with Windows, as Windows will often overwrite the boot-loader (thus wiping out your option to boot Linux) when certain updates are installed. Windows always assumes it's the only operating system in use, so it has no problem wiping out other operating systems that share the same machine.

To use this utility, first boot the affected machine from the Ubuntu live media (the desktop version) and connect to the internet. Then, you can install the Boot Repair utility with the following commands:

```
| sudo add-apt-repository ppa:yannubuntu/boot-repair  
| sudo apt update && apt install boot-repair
```

Now, the Boot Repair utility is installed in the live system. You can access it from the Ubuntu dash, or you can simply enter the `boot-repair` command from the terminal to open it:



The main screen of the Boot Repair utility

For most situations, you can simply choose the default option, Recommended repair. There are advanced options as well if you'd like to do something different; you can view these options by expanding the Advanced options menu. This will allow you to change where GRUB is reinstalled, if you're restoring GRUB onto a different drive than the main drive. It also allows you to add additional options to GRUB if you're having issues with certain hardware configurations, and even allows you to upgrade GRUB to

its most recent version. Using Boot Repair, you should be able to resurrect an unbootable system if the underlying GRUB installation has become damaged.

As you can see, 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, and I'm even more thrilled to have had the opportunity to update this body of work for 18.04. 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 even had the opportunity to write had it not been for you 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.

Questions

1. List some of the ways you can prevent disasters in your organization
2. What is the principle of least privilege, and how can it assist your organization?
3. What utility can be used to keep track of changes done to configuration files on your server?
4. Name some backup strategies you can implement to strengthen your backup scheme
5. Which utility can give you useful information about your drives?
6. The _____ utility can help you in determining which disk(s) were newly added to your server
7. What are some of the things you can use bootable recovery for in regard to system recovery?
8. The _____ command shows you a list of LVM logical volumes on your system

Further Reading

- Pro Git book: <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>

Using the Alternate Installer

In the very first chapter, we worked through installing Ubuntu Server. During that chapter, we used the default installer that is downloaded from the Ubuntu website. This installer is best for most scenarios, but an alternate installer exists for specialized installations. The alternate installer allows you to set up things such as LVM and RAID, and even allows you to manually partition as well. If you've used previous versions of Ubuntu Server, the alternate installer may seem familiar to you, and that's because it's the installer that's been used by default in previous versions (Ubuntu Server versions prior to 18.04). In case you may benefit from an installation walkthrough using the alternate installer, this appendix will guide you through it step by step.

In this appendix, we will cover:

- Obtaining the Alternate Installation Media
- Installing via the Alternate Installer
- Setting up Software RAID

Obtaining the Alternate Installer

As with the default installer, the alternate installer is downloaded in the form of an ISO image that we can use to create a bootable CD/DVD or flash drive. To download the alternate installer, visit the following website: <http://cdimage.ubuntu.com/releases/18.04/release/>

The file you're looking for should be named `ubuntu-18.04-server-amd64.iso` or similar (if there's a point release for example, it may be named `ubuntu-18.04.x-server-amd64.iso` where `x` is the point release number). The ISO image you download can be used in a virtual machine right away, or you can create a bootable media with which to install it on a physical machine. The process of creating a bootable flash drive was covered in [Chapter 1](#), Deploying Ubuntu Server, so refer back to that if you need a refresher on using Etcher to create bootable install media.

Once you have the bootable media created, you're ready to begin. You'll boot your server with this media just as you would with the normal installer; the only difference in set up is which ISO image you're using.

Installing via the Alternate Installer

Now, we'll get on with an actual installation utilizing the alternate installer. At this point, you'll boot your server with the bootable media (flash drive, CD/DVD, and so on) and we'll be off to the races. Go through the following steps to complete the process:

1. First, you'll be asked to select your language, which defaults to English. Choose your language here and press Enter:

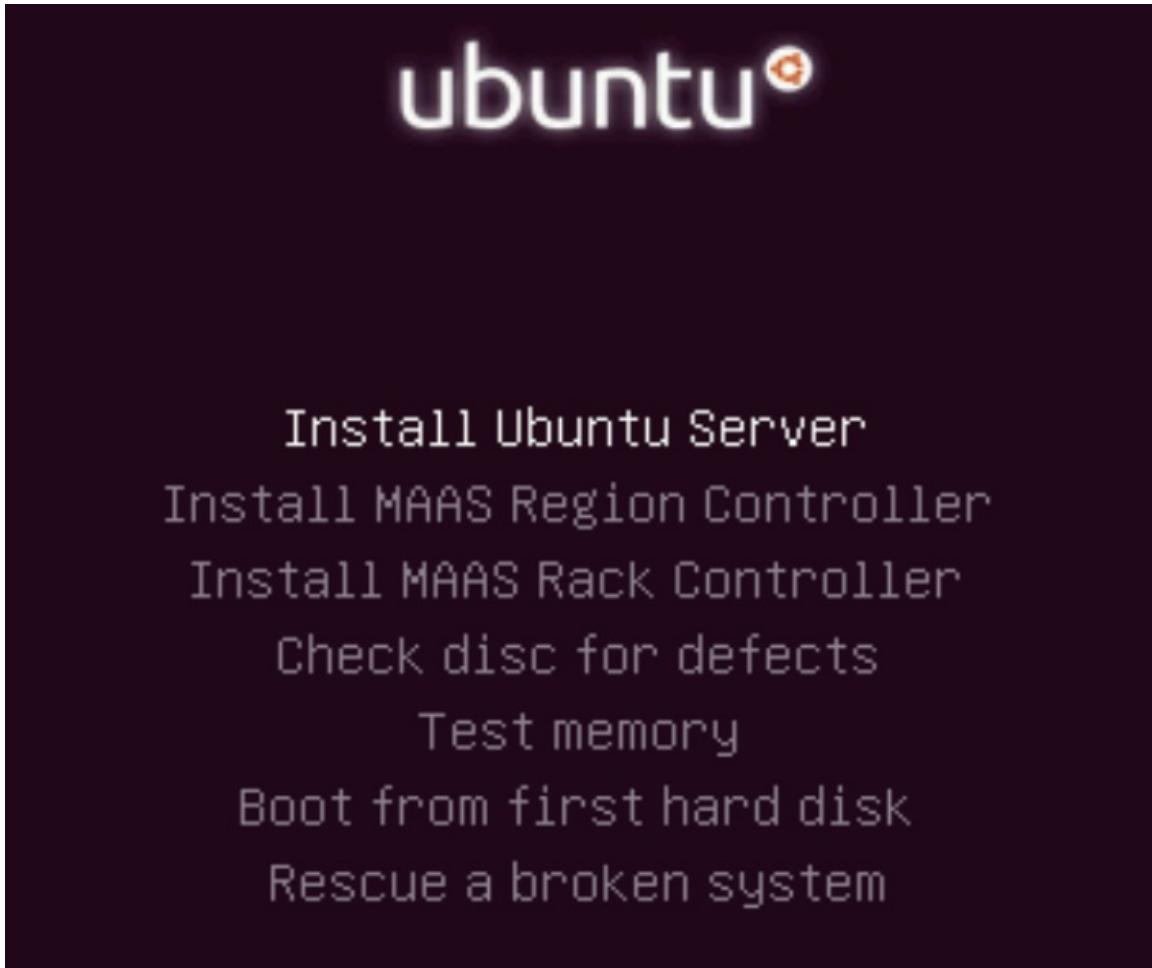
Language			
Amharic	Français	Македонски	Tamil
Arabic	Gaeilge	Malayalam	ଓଡ଼ିଆ
Asturianu	Galego	Marathi	Thai
Беларуская	Gujarati	Burmese	Tagalog
Български	לְבִיָּה	Nepali	Türkçe
Bengali	Hindi	Nederlands	Uyghur
Tibetan	Hrvatski	Norsk bokmål	Українська
Bosanski	Magyar	Norsk nynorsk	Tiếng Việt
Català	Bahasa Indonesia	Punjabi (Gurmukhi)	中文(简体)
Čeština	Íslenska	Polski	中文(繁體)
Dansk	Italiano	Português do Brasil	
Deutsch	日本語	Português	
Dzongkha	ཇାନ୍ମଙ୍ଗୋ	Română	
Ελληνικά	Қазақ	Русский	
English	Khmer	Sámegillii	
Esperanto	ಕನ್ನಡ	ଓଡ଼ିଆ	
Español	한국어	Slovenčina	
Eesti	Kurdî	Slovenščina	
Euskara	Lao	Shqip	
فارسی	Lietuviškai	Српски	
Suomi	Latviski	Svenska	

F1 Help F2 Language F3 Keymap F4 Modes F5 Accessibility F6 Other Options

Selecting your language

2. Next, the main menu for the installer will appear. You can simply press Enter as the default selection is what we want, Install Ubuntu Server.

There are other options that are useful here, especially the memory test. If your server is a physical server, it's a good idea to check the RAM every once in a while for defects. The Rescue a broken system option is useful for troubleshooting.



The main installation menu for the alternate installer

3. Next, you'll select the language that will be used for the remainder of the installer, defaulting to English. If your language is something different, choose that here.



For this screen (and all the remaining sections), you can use your arrow keys to make a selection, Enter to confirm the selection, Tab to change fields, and spacebar to select check boxes.

C	- No localization
Albanian	- Shqip
Arabic	- ئۇرۇغۇ
Asturian	- Asturianu
Basque	- Euskara
Belarusian	- <91>ела<80>
Bosnian	- Bosanski
Bulgarian	- <91><8A>лга
Catalan	- Català
Chinese (Simplified)	- 中<96><87>(
Chinese (Traditional)	- 中<96><87>(
Croatian	- Hrvatski
Czech	- <8C>eština
Danish	- Dansk
Dutch	- Nederlands
English	- English
Esperanto	- Esperanto
Estonian	- Eesti
Finnish	- Suomi
French	- Français
Galician	- Galego
German	- Deutsch
Greek	- <95>λληνι

Setting the language for the installer

4. Next, you'll select your location. When you've highlighted the appropriate selection, press Enter:

[!!] Select your location

The selected location will be used to set your time zone and also for example to help select the system locale. Normally this should be the country where you live.

This is a shortlist of locations based on the language you selected. Choose "other" if your location is not listed.

Country, territory or area:

- Antigua and Barbuda
- Australia
- Botswana
- Canada
- Hong Kong
- India
- Ireland
- Israel
- New Zealand
- Nigeria
- Philippines
- Singapore
- South Africa
- United Kingdom
- United States**
- Zambia
- Zimbabwe
- other

[<Go Back>](#)

Setting your location

5. Next, you'll be given an option to have the system detect your keyboard layout. I think it's faster to decline this and say No. At the screen following this, we can select this ourselves:

[!] Configure the keyboard

You can try to have your keyboard layout detected by pressing a series of keys. If you do not want to do this, you will be able to select your keyboard layout from a list.

Detect keyboard layout?

<Go Back>

<Yes>

<No>

Choosing whether to detect the keyboard layout

6. On the next screen, select the country of origin for your keyboard:

[!] Configure the keyboard

The layout of keyboards varies per country, with some countries having multiple common layouts. Please select the country of origin for the keyboard of this computer.

Country of origin for the keyboard:

- Bambara
- Bangla
- Belarusian
- Belgian
- Berber (Algeria, Latin)
- Bosnian
- Braille
- Bulgarian
- Burmese
- Chinese
- Croatian
- Czech
- Danish
- Dhivehi
- Dutch
- Dzongkha
- English (Australian)
- English (Cameroon)
- English (Ghana)
- English (Nigeria)
- English (South Africa)
- English (UK)
- English (US)**

[<Go Back>](#)

Setting the country of origin for your keyboard

7. Next, select your actual keyboard layout:

[!] Configure the keyboard

Please select the layout matching the keyboard for this machine.

Keyboard layout:

English (US)

English (US) - Cherokee

English (US) - English (Colemak)

English (US) - English (Dvorak)

English (US) - English (Dvorak, alt. intl.)

English (US) - English (Dvorak, intl., with dead keys)

English (US) - English (Dvorak, left-handed)

English (US) - English (Dvorak, right-handed)

English (US) - English (Macintosh)

English (US) - English (US, alt. intl.)

English (US) - English (US, euro on 5)

English (US) - English (US, intl., with dead keys)

English (US) - English (Workman)

English (US) - English (Workman, intl., with dead keys)

English (US) - English (classic Dvorak)

English (US) - English (intl., with AltGr dead keys)

English (US) - English (programmer Dvorak)

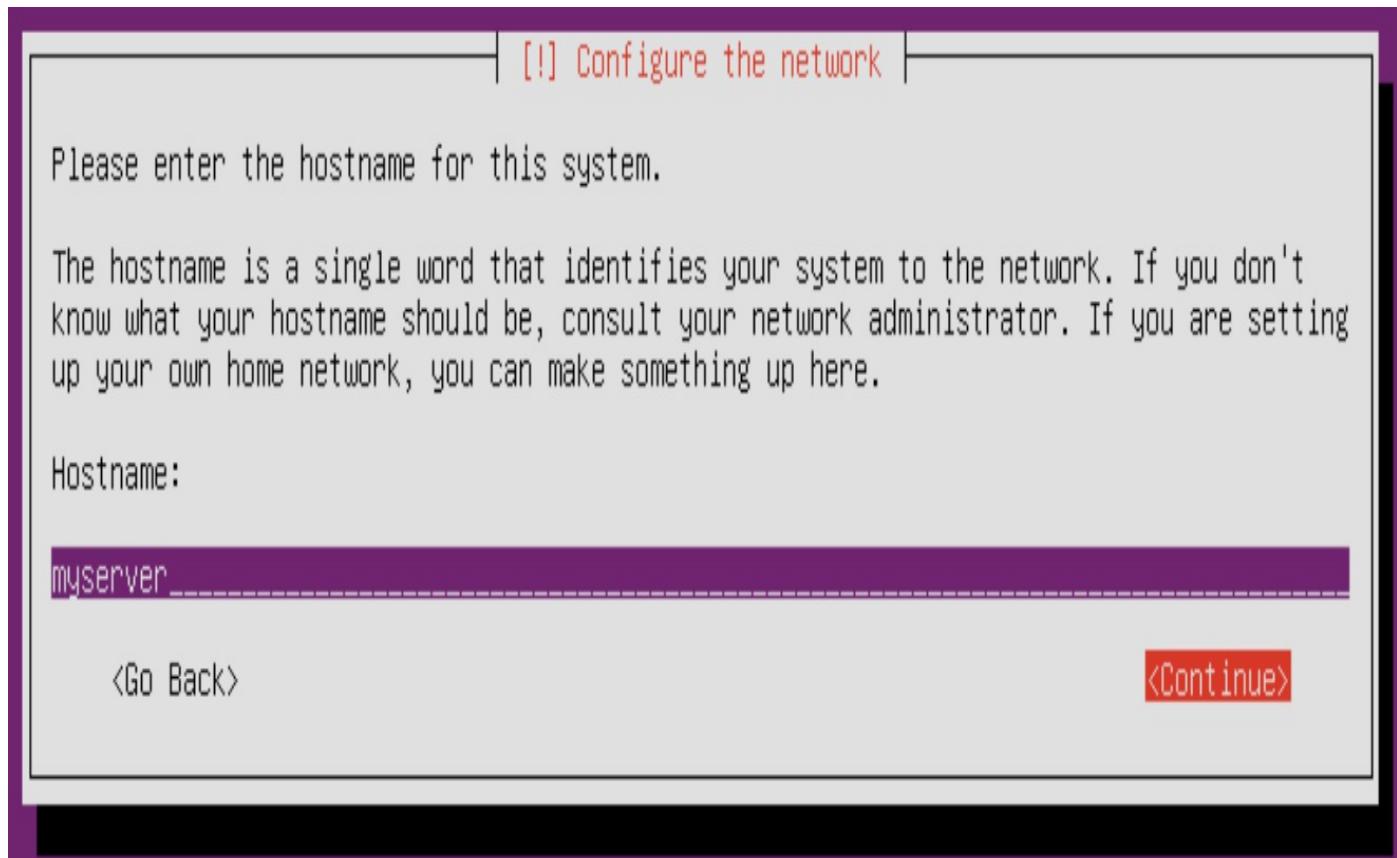
English (US) - English (the divide/multiply keys toggle the layout)

English (US) - Russian (US, phonetic)

English (US) - Serbo-Croatian (US)

<Go Back>

8. After a few progress bars scroll by, the installer will ask you to name your server. This name will appear on your network if you share files from the server, as well as your command prompt. After you type a name, press Enter:



Setting the host name of your system

9. Now, you'll enter a name for your user account. This is the display name, not the name that will be used at login. When you've finished typing your name, press Enter:

[!!] Set up users and passwords

A user account will be created for you to use instead of the root account for non-administrative activities.

Please enter the real name of this user. This information will be used for instance as default origin for emails sent by this user as well as any program which displays or uses the user's real name. Your full name is a reasonable choice.

Full name for the new user:

Jay LaCroix

<Go Back>

<Continue>

Setting the user's full name

10. Create a login name for your user. This is the user that will be given access to `sudo` by default. After you've typed your desired username, press Enter:

[!!] Set up users and passwords

Select a username for the new account. Your first name is a reasonable choice. The username should start with a lower-case letter, which can be followed by any combination of numbers and more lower-case letters.

Username for your account:

jay

<Go Back>

<Continue>

Adding a password for the user

11. Next, create a super-secure password for your user and press Enter:

[!!] Set up users and passwords

A good password will contain a mixture of letters, numbers and punctuation and should be changed at regular intervals.

Choose a password for the new user:

[] Show Password in Clear

<Go Back>

<Continue>

Setting the username

12. Verify the password by entering it again, then press Enter:

[!!] Set up users and passwords

Please enter the same user password again to verify you have typed it correctly.

Re-enter password to verify:

[] Show Password in Clear

<Go Back>

<Continue>

Confirming the new user's password

13. Next, we'll set the time zone. The default time should work be correct, but we can always change this later. Press Tab key to select Yes, and then press Enter:

[!] Configure the clock

Based on your present physical location, your time zone is America/New_York.

If this is not correct, you may select from a full list of time zones instead.

Is this time zone correct?

<Go Back>

<Yes>

<No>

Confirming time zone selection

14. At this point, we have a choice. We can select Guided - use entire disk, which is the desired choice for most as it gives you a basic install. If you would like to set up RAID, choose Manual. If you do choose Manual to set up RAID, take a detour to the next section where I will walk you through the process. Once done, come back here and follow along with the remaining steps.

[!!] Partition disks

The installer can guide you through partitioning a disk (using different standard schemes) or, if you prefer, you can do it manually. With guided partitioning you will still have a chance later to review and customise the results.

If you choose guided partitioning for an entire disk, you will next be asked which disk should be used.

Partitioning method:

- Guided - use entire disk**
- Guided - use entire disk and set up LVM
- Guided - use entire disk and set up encrypted LVM
- Manual

<Go Back>

Choosing a mode for partitioning

15. Next, we'll select the disk that we want to install Ubuntu onto. If you have only one disk, you'll see only one entry here. Otherwise, select the entry that corresponds to the disk you'll be installing Ubuntu onto. Then, press Enter:

[!!] Partition disks

Note that all data on the disk you select will be erased, but not before you have confirmed that you really want to make the changes.

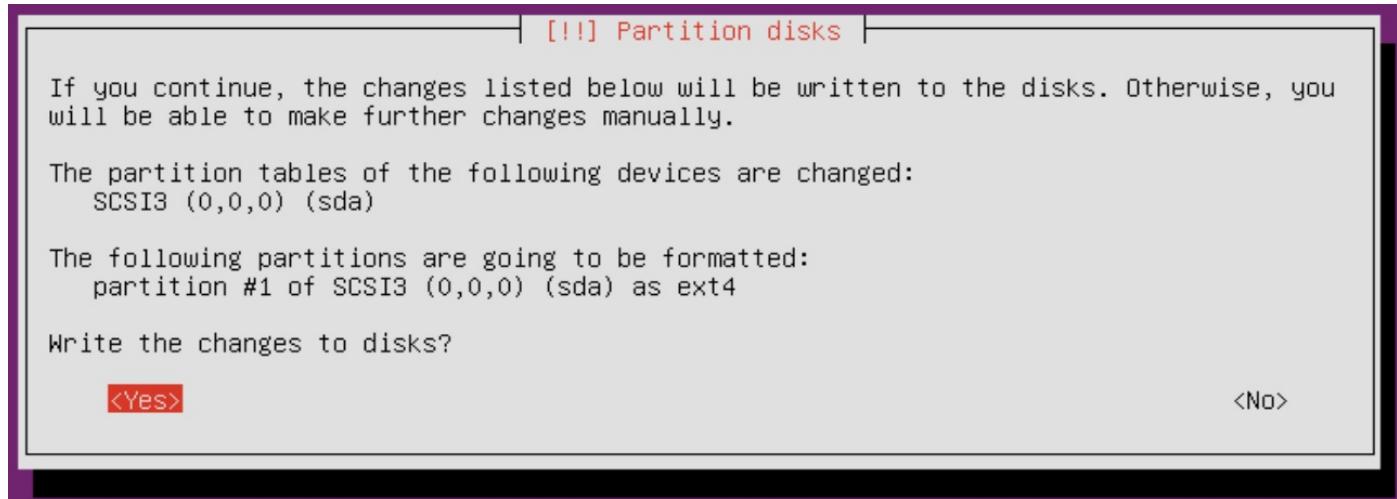
Select disk to partition:

SCSI3 (0,0,0) (sda) - 21.5 GB ATA VBOX HARDDISK

<Go Back>

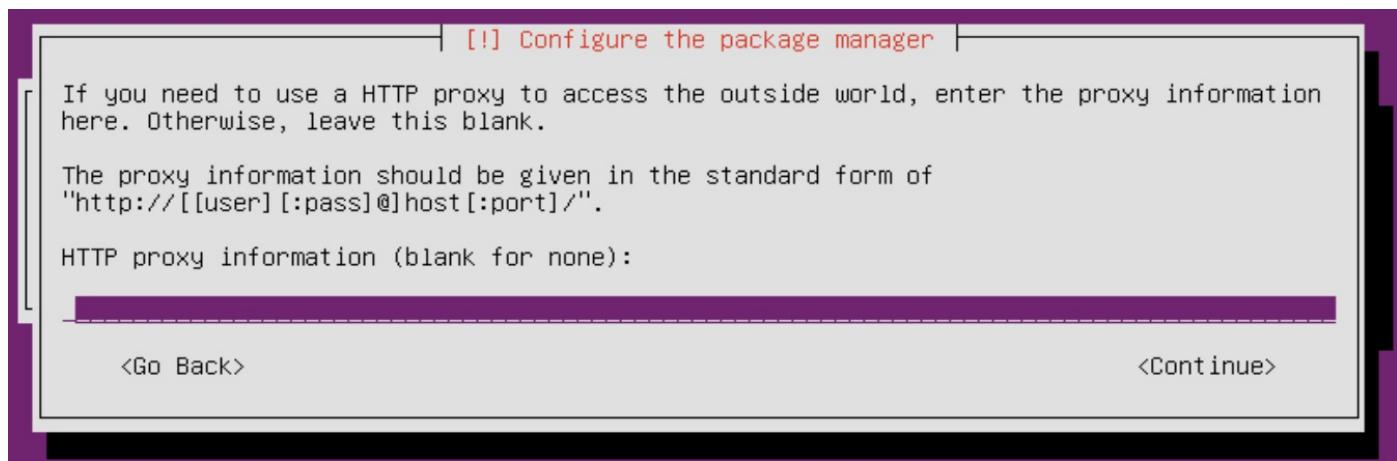
Selecting the desired disk for installation

16. Next, we'll confirm our changes. Press Tab to highlight Yes, and press Enter:



Confirming our changes to the disk

17. Now, we'll be asked if we have a proxy on our network. If you do, enter the information here. Either way, press Enter to go on:



Setting the HTTP proxy if we have one

18. Next, we'll be asked if we want to enable automatic updates. This is optional, and you may want to select No automatic updates for now. If you desire a more secure system, it's probably a good idea to enable automatic updates, but the choice is yours:

```
[!] Configuring tasksel

Applying updates on a frequent basis is an important part of keeping your system secure.

By default, updates need to be applied manually using package management tools.
Alternatively, you can choose to have this system automatically download and install
security updates, or you can choose to manage this system over the web as part of a group
of systems using Canonical's Landscape service.

How do you want to manage upgrades on this system?

No automatic updates
Install security updates automatically
Manage system with Landscape
```

Choosing whether to enable automatic updates

19. On the next screen, we'll have a selection for some useful package sets. Here, we can install the required packages for setting up a DNS server, LAMP server, and so on. The one option I definitely recommend you enable is OpenSSH server. This is, of course, optional. However, most administrators use SSH to configure their servers. Choose Continue to move on:

```
[!] Software selection

At the moment, only the core of the system is installed. To tune the system to your
needs, you can choose to install one or more of the following predefined collections of
software.

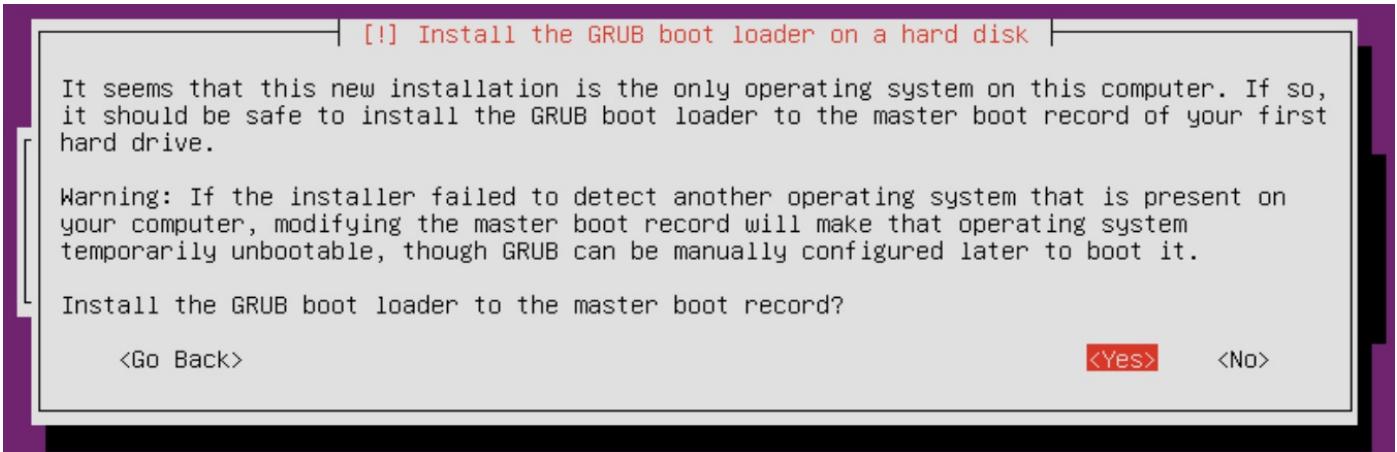
Choose software to install:

[ ] DNS server
[ ] LAMP server
[ ] Mail server
[ ] PostgreSQL database
[ ] Print server
[ ] Samba file server
[*] OpenSSH server

<Continue>
```

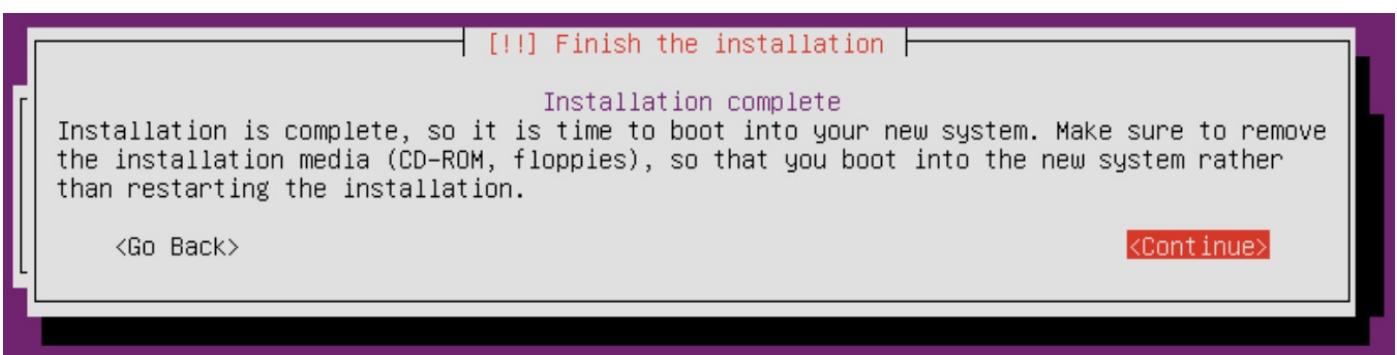
Choosing between optional package sets during installation

20. Next, we'll choose whether or not to install GRUB on the master boot record. You should definitely say yes to this. Press Tab key to highlight Yes, and then press Enter to move on:



Confirming the installation of the GRUB boot loader

21. We did it! That concludes all of the steps. On the final screen, we'll press Enter to end the installation process and reboot the server:

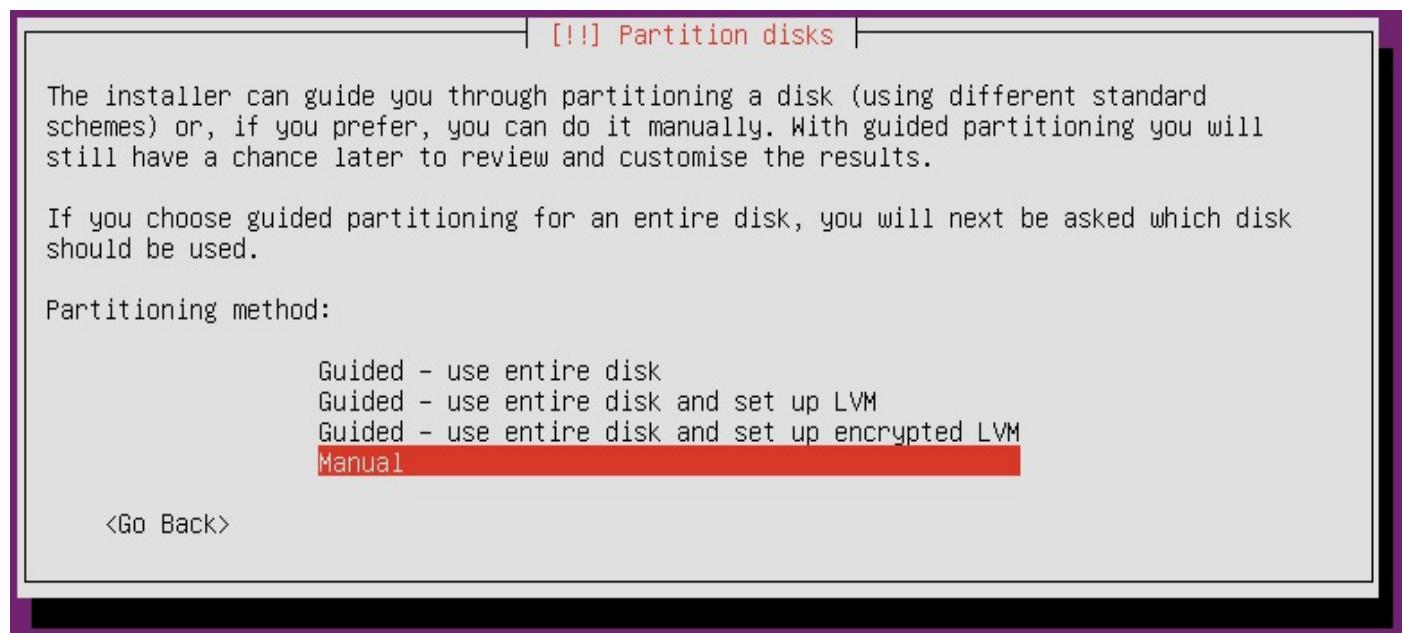


Finishing the installation

Setting up software RAID

Part of the benefit of the alternate installer is that it allows us to install Ubuntu in ways that aren't normally available with the live installer. One of the additional features is to be able to install Ubuntu on software RAID, which is what we will do in this section. Here, the steps that we'll go through actually continue on from step 14 in the previous section. Follow these steps to set up an installation with software RAID, specifically with RAID1 between two disks in this case.

1. In the previous section, we chose Guided - use entire disk at the screen shown in the following screenshot. To set up RAID, we'll select Manual on this screen instead:



Choosing the option to manually partition our system

2. Next, we'll select the first disk and press Enter:

[!!] Partition disks

This is an overview of your currently configured partitions and mount points. Select a partition to modify its settings (file system, mount point, etc.), a free space to create partitions, or a device to initialize its partition table.

Guided partitioning

Configure iSCSI volumes

SCSI3 (0,0,0) (sda) - 21.5 GB ATA VBOX HARDDISK

SCSI4 (0,0,0) (sdb) - 21.5 GB ATA VBOX HARDDISK

Undo changes to partitions

Finish partitioning and write changes to disk

<Go Back>

Selecting our first disk

3. Next, you'll be asked whether you will want to create a new partition table on this disk, which will wipe all data on it. Choose Yes and press Enter:

[!!] Partition disks

You have selected an entire device to partition. If you proceed with creating a new partition table on the device, then all current partitions will be removed.

Note that you will be able to undo this operation later if you wish.

Create new empty partition table on this device?

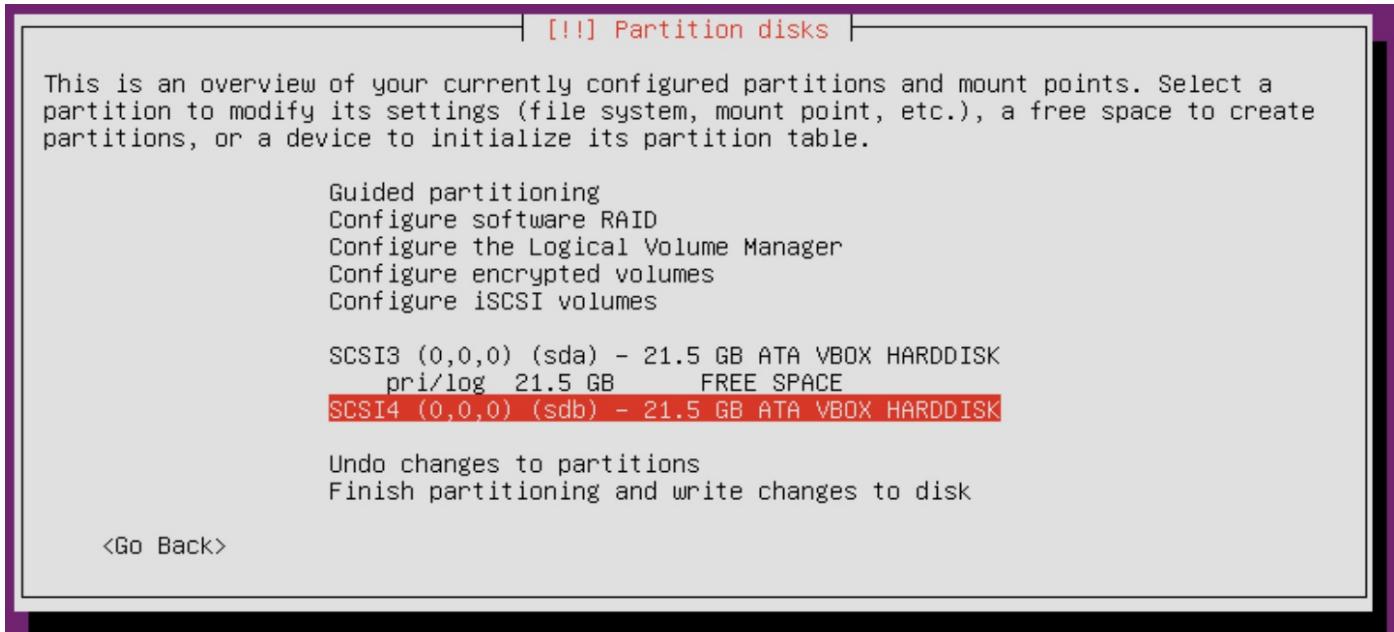
<Go Back>

<Yes>

<No>

Confirming the initialization of the first disk

4. On the next screen, select the second disk (the one we haven't initialized yet) and press Enter:



Selecting the second disk

5. You'll again be asked whether you wish to create a new partition table.
Select Yes and press Enter:



Confirming the creation of a new partition table on the second disk

6. On the next screen, we'll choose the Configure software RAID option:

[!!] Partition disks

This is an overview of your currently configured partitions and mount points. Select a partition to modify its settings (file system, mount point, etc.), a free space to create partitions, or a device to initialize its partition table.

Guided partitioning

Configure software RAID

Configure the Logical Volume Manager

Configure encrypted volumes

Configure iSCSI volumes

SCSI3 (0,0,0) (sda) - 21.5 GB ATA VBOX HARDDISK

 pri/log 21.5 GB FREE SPACE

SCSI4 (0,0,0) (sdb) - 21.5 GB ATA VBOX HARDDISK

 pri/log 21.5 GB FREE SPACE

Undo changes to partitions

Finish partitioning and write changes to disk

<Go Back>

Choosing the software RAID option

7. Before we can continue, the installer must finalize the changes we've made so far (which at this point has only been to initialize the disks). Select Yes and press Enter:

[!!] Partition disks

Before RAID can be configured, the changes have to be written to the storage devices. These changes cannot be undone.

When RAID is configured, no additional changes to the partitions in the disks containing physical volumes are allowed. Please convince yourself that you are satisfied with the current partitioning scheme in these disks.

The partition tables of the following devices are changed:

SCSI3 (0,0,0) (sda)

SCSI4 (0,0,0) (sdb)

Write the changes to the storage devices and configure RAID?

<Yes>

<No>

Finalizing disk initialization

8. Next, choose Create MD device and press Enter:

[!!] Partition disks

This is the software RAID (or MD, "multiple device") configuration menu.

Please select one of the proposed actions to configure software RAID.

Software RAID configuration actions

Create MD device

Delete MD device

Finish

<Go Back>

Getting ready to create a new MD device

9. Next, we'll select the type of RAID we wish to work with. Select RAID1 and press Enter to continue:

[!!] Partition disks

Please choose the type of the software RAID device to be created.

Software RAID device type:

- RAID0
- RAID1
- RAID5
- RAID6
- RAID10

[<Go Back>](#)

Selecting the type of RAID we will use

10. Next, choose the number of disks we will add to RAID1. RAID1 requires exactly two disks. Enter `2` here and press Enter:

[!!] Partition disks

The RAID1 array will consist of both active and spare devices. The active devices are those used, while the spare devices will only be used if one or more of the active devices fail. A minimum of 2 active devices is required.

NOTE: this setting cannot be changed later.

Number of active devices for the RAID1 array:

`2`

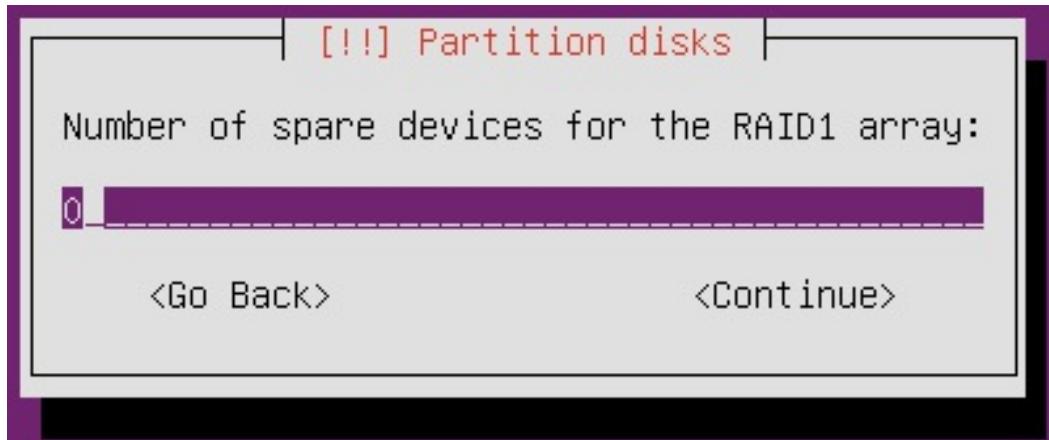
[<Go Back>](#)

[<Continue>](#)

Choosing the number of disks we will use

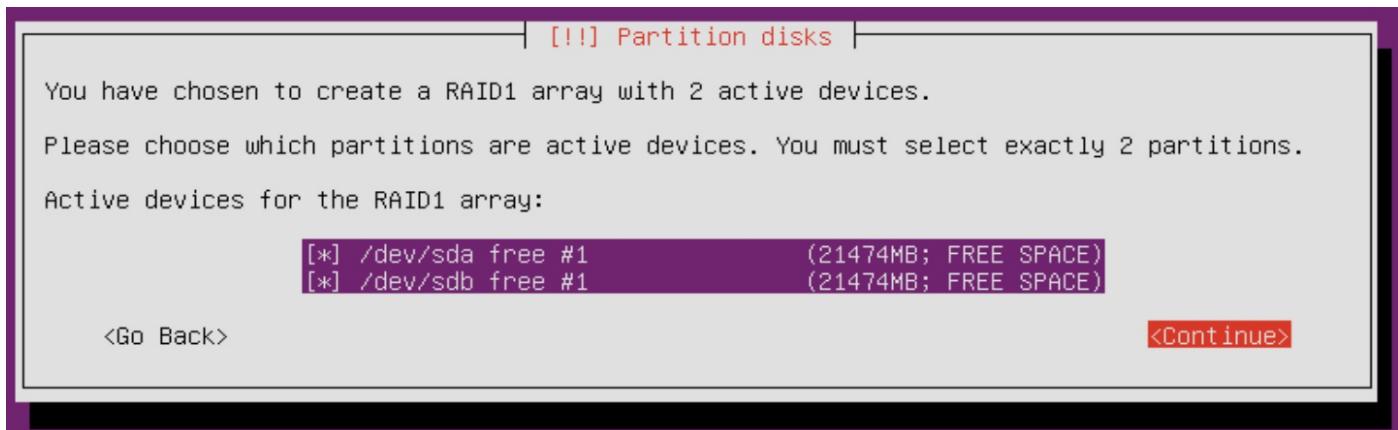
11. If we have additional drives beyond two, we can add one or more hot spares here. These will be used in the event of a failure with one of the RAID disks. If you have

an extra disk, feel free to use it and enter the appropriate number here. Either way, select Continue and press Enter:



Choosing how many hot spares to initialize

12. Next, we will choose which disks to include with our RAID1 configuration. We should have exactly two listed. Use the arrow keys to move between them, and press Space to select a disk. We need to select both disks, which means we will be marking both with an asterisk (*). Once you've selected both disks, select Continue and press Enter:



Choosing the disks to use with the RAID configuration

13. Next, we'll finalize our selections so far. Select Yes and press Enter:

[!!] Partition disks

Before RAID can be configured, the changes have to be written to the storage devices. These changes cannot be undone.

When RAID is configured, no additional changes to the partitions in the disks containing physical volumes are allowed. Please convince yourself that you are satisfied with the current partitioning scheme in these disks.

The partition tables of the following devices are changed:

SCSI3 (0,0,0) (sda)
SCSI4 (0,0,0) (sdb)

Write the changes to the storage devices and configure RAID?

<Yes>

<No>

Finalize RAID changes

14. Select Finish to continue on:

[!!] Partition disks

This is the software RAID (or MD, "multiple device") configuration menu.

Please select one of the proposed actions to configure software RAID.

Software RAID configuration actions

Create MD device
Delete MD device
Finish

<Go Back>

Selecting Finish to finish setting up RAID

15. Now, we have successfully set up RAID. However, that alone isn't enough; we need to also format it and give it a mount point. Use the arrow keys to select the RAID device we just set up and press Enter:

```
[!!] Partition disks

This is an overview of your currently configured partitions and mount points. Select a
partition to modify its settings (file system, mount point, etc.), a free space to create
partitions, or a device to initialize its partition table.

    Guided partitioning
    Configure software RAID
    Configure the Logical Volume Manager
    Configure encrypted volumes
    Configure iSCSI volumes

    RAID1 device #0 - 21.5 GB Software RAID device
        #1      21.5 GB
        SCSI3 (0,0,0) (sda) - 21.5 GB ATA VBOX HARDDISK
            #1 primary 21.5 GB   K raid
        SCSI4 (0,0,0) (sdb) - 21.5 GB ATA VBOX HARDDISK
            #1 primary 21.5 GB   K raid

    Undo changes to partitions
    Finish partitioning and write changes to disk

<Go Back>
```

Selecting our new RAID device so we can configure it

16. Using the following screenshot as a guide, make sure you set Use as to EXT4 journaling file system (you can choose another filesystem type if you wish to experiment). In addition, set the Mount point to / and then select Done setting up the partition:

```
[!!] Partition disks

You are editing partition #1 of RAID1 device #0. No existing file system was detected in
this partition.

Partition settings:

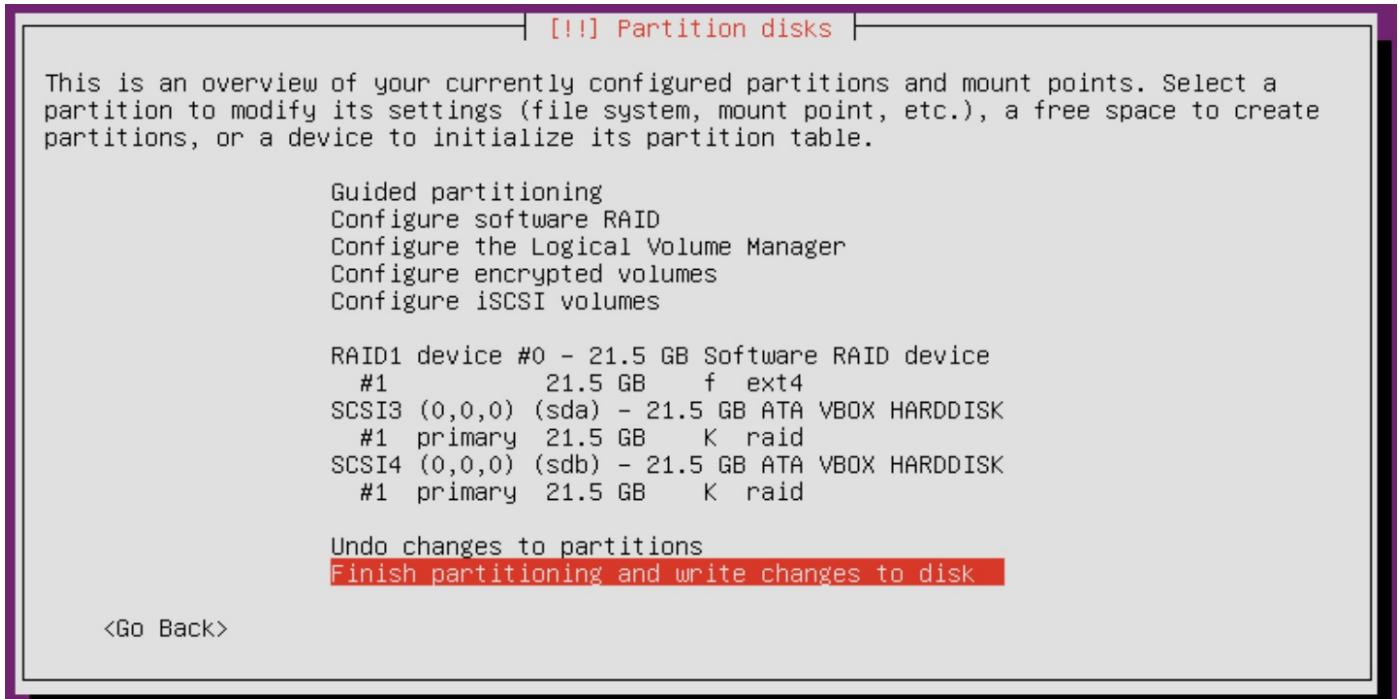
    Use as:          Ext4 journaling file system
    Mount point:     /
    Mount options:   defaults
    Label:           none
    Reserved blocks: 5%
    Typical usage:   standard

    Erase data on this partition
    Done setting up the partition

<Go Back>
```

Reviewing configuration settings for our RAID device

17. To finalize our selection, choose Finish partitioning and write changes to disk:



Finishing partitioning

18. Next, we'll see one last confirmation before our new partition will be created. Select Yes and press Enter:



Final confirmation for this portion of the installation routine

That pretty much covers the process of setting up RAID. From here, you can return to the previous section and continue on with the installation process with step 16. Once you finish the entire installation process, it will be useful to know how to check the health of the RAID. Once you log in to your new installation, execute the following command:

```
| cat /proc/mdstat
```

```
jay@ubuntu:~  
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:~$
```

Checking the health of the RAID device

In the previous screenshot, you can see that we're using RAID1 (look for the section of the output that reads active raid1). We can also see whether each disk in the RAID array is online, looking for the appropriate number of Us. It's this line: 20953088 blocks super 1.2 [2/2] [UU]

If the output included U_ or _U instead of UU, we would have cause for alarm, because that would mean one of the disks encountered an issue or is offline. Basically, each U represents a disk, and each underscore represents where a disk should be. If one is missing and changed to an underscore, we have a problem with our RAID array.

Summary

In this extra chapter, we walked through the process of not only using the alternate installer, but installing Ubuntu Server with RAID as well. When it comes to physical servers, having an extra disk can help keep us online if a disk fails. While RAID is certainly not to be considered a solution for backups, having an extra safety net in place will only benefit us. RAID is definitely something to consider if we wish to set up a more resilient server.

Assessments

Chapter 1 – Deploying Ubuntu Server

1. Role
2. ISO
3. Acceptable answers include Digital Ocean, Linode, or any other valid VPS provider
4. Encryption at Rest (LUKS is also an acceptable answer)
5. LTS stands for Long-Term Support. LTS releases are supported for a longer period of time than non-LTS releases.
6. Etcher
7. The root filesystem ("/" is also an acceptable answer)

Chapter 2 – Managing Users

1. sudo
2. adduser (useradd is also acceptable)
3. userdel
4. /etc/passwd and /etc/shadow
5. /etc/skel
6. sudo adduser jdoe (useradd is also acceptable)
7. sudo groupadd accounting
8. visudo
9. chmod, chown

Chapter 3 – Managing Storage Volumes

1. du
2. Filesystem Hierarchy Standard
3. df
4. mount
5. fdisk
6. /etc/fstab
7. swap
8. lvdisplay

Chapter 4 – Connecting to Networks

1. Hostname
2. ip addr show
3. Netplan
4. OpenSSH
5. Passphrase
6. Static
7. etc/nsswitch.conf

Chapter 5 – Managing Software Packages

1. apt
2. Hardware Enablement updates give you a new kernel with updated drivers that can support newer hardware
3. apt install <package>
4. apt-remove <package>
5. The snap refresh command will update Snap packages installed on your server
6. Debian and Snap packages
7. aptitude
8. A package repository is an online resource that provides packages to Ubuntu systems

9. A Personal Package Archive (PPA) is like a normal package repository, but typically serves a small purpose rather than being a resource for a large group of packages.
10. dpkg --get-selections > packages.list
11. sudo apt autoremove
12. Snap packages are universal, so they can be installed on any distribution with Snap support. In addition, you can receive newer versions of packages. Finally, they don't conflict with system packages.

Chapter 6 – Controlling and Monitoring Processes

1. Ps
2. jobs
3. kill
4. htop
5. systemctl
6. free
7. crontab -e

Chapter 7 – Setting Up Network Services

1. Subnetting is the process of splitting up your network into smaller networks. In our case, it allows us to have a larger number of IP addresses to assign via DHCP.
2. isc-dhcp-server
3. /var/lib/dhcp/dhcpd.leases
4. A DNS server maps names to IP addresses
5. If the primary DNS server goes down, the secondary would allow for name resolution to continue to happen
6. dig
7. NTP

Chapter 8 - Accessing and sharing files

1. Samba, NFS
2. SSHFS
3. scp, rsync
4. Samba is a good choice in mixed environments, where you have both Windows and Linux clients
5. NFS is a good choice in UNIX/Linux environments, and where permissions are important
6. /etc/idmapd.conf
7. fusermount -u <directory>

Chapter 9 - Sharing and Transferring Files

1. MySQL
2. mysql_secure_installation
3. CREATE USER 'samus'@'10.100.1.%' IDENTIFIED BY 'sr388';
4. GRANT ALL ON MyAppDB.* TO 'william'@'192.168.1.%' IDENTIFIED BY 'mysecretpassword';
5. mysqldump
6. SELECT HOST, USER, PASSWORD FROM mysql.user;
7. DROP DATABASE MyAppDB;

Chapter 10 – Serving Web Content

1. Apache and NGINX
2. apache2
3. Virtual Hosts
4. a2ensite
5. Self-signed certificates
6. Document Root
7. 443
8. priority

Chapter 11 – Learning Advanced Shell Techniques

1. Zsh, Ksh, Fish, and so on
2. alias
3. Hashbang
4. history
5. Standard Output
6. Standard Error
7. >>
8. >
9. \$

Chapter 12 – Virtualization

1. KVM is built into the Linux kernel, while QEMU allows for hardware emulation
2. virt-manager
3. Storage Pool
4. Bridge
5. Install the latest updates, regenerate the OpenSSH host keys
6. The CPU must support virtualization extensions and they must be enabled
 - The required libvirt packages need to be present
 - The libvirtd service needs to be running
 - The user must have access to the server via SSH
 - The user must be a member of the appropriate group

Chapter 13 – Running Containers

1. Docker, LXD
2. Containers are lighter in resource usage than virtual machines. However, not all applications are capable of running in a container.
3. The Docker Hub
4. LXC

5. Canonical
6. Dockerfile
7. Storage pools, snapshots, live migration

Chapter 14 – Automating Server Configuration with Ansible

1. Chef, Puppet, Salt Stack
2. It's fast, light on resources, and doesn't require an agent on its targets
3. Inventory
4. OpenSSH
5. Playbook, plays
6. Apt, copy, service
7. Acceptable answers include (but are not limited to) file, user, group, or any other valid Ansible module
8. The inventory method is useful in a static environment, and the pull method is best for dynamic environments

Chapter 15 – Securing Your Server

1. This stands for Common Vulnerabilities and Exposures, and is a database containing lists of known exploits and vulnerabilities for a given platform, along with scope of the issue and mitigation steps (if known)
2. Livepatch
3. jail.conf, jail.local
4. Disable password authentication, disable root login, place it behind a firewall, change the default port, set up an allowed users or groups list
5. The principle of least privilege refers to giving users only the permissions to resources they absolutely need for their job, thus limiting the scope if they make a mistake

6. netstat -tulpn
7. cryptsetup
8. Custom graphs, hardware information, reports, or any other feature not mentioned

Chapter 16 – Troubleshooting Ubuntu Servers

1. Bash history allows an administrator to see a list of commands recently executed on the server, which comes in handy when looking at what was done to solve an issue in the past or trying to cross-reference recent issues to recently executed commands
2. A successful root cause analysis assists in the troubleshooting process as it can provide an overview of the underlying issue or initial cause of disruption
3. A description of the issue, the related application or hardware, what was discovered during the process, and what was done to resolve it
4. /var
5. head, tail, less
6. nslookup, dig, host
7. spci

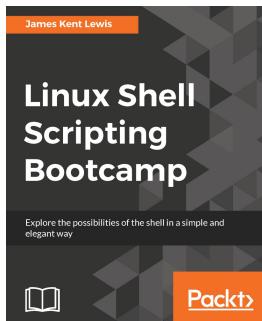
Chapter 17 – Preventing and Recovering from Disasters

1. Successful answers include, but are not limited to, coming up with a recovery plan, backup, and so on
2. This principle limits what users are able to do, thus limiting the scope of damage they can cause
3. Git

4. No one right answer here, but some good ideas are testing backups, coming up with a good plan, and taking advantage of incremental backups
5. hdparm
6. fdisk
7. Reinstalling the boot loader, recovering files, and other acceptable answers

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Linux Shell Scripting Bootcamp

James Kent Lewis

ISBN: 978-1-78728-110-3

- Get acquainted with the basics of a shell script to serve as a refresher for more advanced topics
Learn different ways to create and run a script
- Discuss the passing and verification of parameters, along with the verification of other items.
- Understand the different forms of conditions and loops, and go over the sleep command in detail
- Learn about different ways to handle the reporting of return codes
- Learn about different ways to handle the reporting of return codes
- Create an interactive script by reading the keyboard and use subroutines and interrupts
- Create scripts to perform backups and go over the use of encryption tools and checksums
- Use wget and curl in scripts to get data directly from the Internet



Linux Device Drivers Development

John Madieu

ISBN: 978-1-78528-000-9

- Use kernel facilities to develop powerful drivers
- Develop drivers for widely used I2C and SPI devices and use the regmap API
- Write and support devicetree from within your drivers
- Program advanced drivers for network and frame buffer devices
- Delve into the Linux irqdomain API and write interrupt controller drivers
- Enhance your skills with regulator and PWM frameworks
- Develop measurement system drivers with IIO framework
- Get the best from memory management and the DMA subsystem
- Access and manage GPIO subsystems and develop GPIO controller drivers

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!