DevOps for the Desperate

A Hands-On Survival Guide

BRADLEY SMITH

no starch press®

DEVOPS FOR THE DESPERATE

A Hands-on Survival Guide

Bradley Smith

no starch press®

San Francisco

Second printing

27 26 25 24 23 2 3 4 5 6

ISBN-13: 978-1-7185-0248-2 (print)
ISBN-13: 978-1-7185-0249-9 (ebook)

Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Editor: Paula Williamson
Developmental Editor: Jill Franklin
Cover Illustration: Gina Redman
Interior Design: Octopod Studios
Technical Reviewer: Quentin Hartman
Copyeditor: Doug McNair
Compositor: Happenstance Type-O-Rama
Proofreader: Jamie Lauer

Library of Congress Cataloging-in-Publication Data

```
Names: Smith, Bradley (Software engineer), author.
Title: DevOps for the desperate : a hands-on survival guide / Bradley
   Smith.
Description: San Francisco, CA : No Starch Press, Inc., [2022] | Includes
   index.
Identifiers: LCCN 2021060922 (print) | LCCN 2021060923 (ebook) | ISBN
   9781718502482 (paperback) | ISBN 9781718502499 (ebook)
Subjects: LCSH: Computer software--Development--Management. | Software
   engineering--Management.
Classification: LCC QA76.76.D47 S567 2022  (print) | LCC QA76.76.D47
   (ebook) | DDC 005.1068--dc23/eng/20220111
LC record available at https://lccn.loc.gov/2021060922
LC ebook record available at https://lccn.loc.gov/2021060923
```

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

[E]

This book is for all the engineers
slogging through on call.

About the Author

Bradley Smith is a director of infrastructure and resides in Denver, Colorado. He has been an engineer for more than 20 years at many startups and businesses, large and small. He has built, trained, and been a member of numerous DevOps, SRE, and software engineering teams. A Boston native, Bradley graduated from the University of Massachusetts Lowell.

About the Technical Reviewer

Quentin Hartman has been living and breathing DevOps since before it had a name. He loves the tech, but more than that, he loves seeing how DevOps practices make software and the lives of people who build it better. Over the course of his nearly 25-year career in technology, Quentin has worked in public education, higher education, nonprofits, and private businesses with anywhere from 3 to 300,000 employees. He has managed telecom systems, datacenters, and public and private clouds. He has acted as a sysadmin, a DBA, a network engineer, an incident responder, and a leader. This broad experience has given him an especially strong foundation in DevOps, which has been his primary focus since 2012. Wherever Quentin is, he puts people before tech and is only really happy when he's working on a social-impact mission using open source tools. Quentin lives near Denver, Colorado, with his family. He can often be found building things, cooking, and wandering in the woods. He can be reached as qhartman on many platforms, including Mastodon.social, Twitter, and LinkedIn.

BRIEF CONTENTS

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

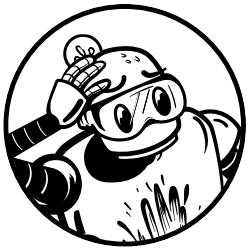
When writing acknowledgments, you quickly realize how many people make publishing a book possible. This would be a very long section if I thanked everyone who contributed in some way, and since this is not a Nobel Prize acceptance speech, I will try to keep it short and sweet. If I do not mention you below, please know I appreciate your help tremendously.

First, I want to thank everyone at No Starch Press. Without you, this book would not have been possible. The guidance from my editor, Jill Franklin, and technical editors, Kyle Terrien and Quentin Hartman, has been invaluable. Thank you so much for wrangling this idea into a book. I appreciate you all.

We all need help from our friends, and this book has my friends' fingerprints all over it. Many of you provided feedback, and I thank you all so much. In particular, I want to thank Rishi Malik, Jaden Grossman, and Jeffrey Matthias. You provided support and (more importantly) lent me your precious time. I owe you!

Finally, I want to thank my family. Countless times, I asked you to read a sentence or a paragraph and tell me what you thought of it—even though you had no idea what I was talking about. To my wife, Leilani, you have always encouraged me and made me believe I could do this. Thank you for making time in our lives so I could work on this book. To my daughters, Aiden and Akira, you are my inspiration, and you make me want to be the best person I can be. I love the three of you, always.

INTRODUCTION



Every day of their working lives, DevOps engineers immerse themselves in cloud-based trends and technologies. Meanwhile, everyone else in engineering is expected to be familiar with DevOps and keep pace with how it is evolving. The reason is simple: DevOps is an integral part of software development. However, you probably don't have time to both do your day job and keep tabs on the ever-changing landscape of DevOps—and luckily, you don't have to. Just gain an understanding of the foundational concepts, terms, and tactics of DevOps, and you'll go far.

On the other hand, when it comes time to deliver code, you can't just put your head in the sand and hope someone else will deal with it. Writing configuration files, enforcing observability, and setting up continuous integration/continuous delivery (CI/CD) pipelines have become the norm in software development. You therefore need to be well versed in code and infrastructure.

If you're a software engineer, developer, or systems administrator, this book will teach you the concepts, commands, and techniques that will give you a solid foundation in DevOps, reliability, and modern application stacks. But be aware that this is an introduction to DevOps, not a definitive guide. I've chosen to keep the knowledge fire hose turned down low, and I'll focus on the following foundational concepts:

Infrastructure as code

Configuration management

Security

Containerization and orchestration

Delivery

Monitoring and alerting

Troubleshooting

Plenty of other great books will take you on a deep dive into the concepts and culture of DevOps. I encourage you to read them and learn more. But if you just want to get started with the basics, *DevOps for the Desperate* has you covered.

What Is the Current State of DevOps?

Over the past few years, different trends have emerged in DevOps. There is a heavy focus on microservices, container orchestration (Kubernetes), automated code delivery (CI/CD), and observability (detailed logging, tracing, monitoring, and alerting). These topics aren't new to the DevOps community, but they're gaining more attention because everyone has swallowed the red pill and gone down the cloud-and-containerization rabbit hole.

Automating and testing the "code to customer" experience is still one of the most important parts of DevOps, and it will continue to be as late adopters play catch-up. As engineering ecosystems mature, more and more DevOps work is occurring higher up the tech stack. In other words, DevOps engineers are heavily relying on tools and processes so software engineers can self-serve shipping code. Because of this, sharing DevOps practices and techniques with feature teams is paramount to delivering standardized and predictable software.

A few more emerging trends are worth a brief mention here. The first is security. DevSecOps is becoming an essential part of the build process rather than a post-release afterthought. Another trend is the use of machine learning for data-driven decisions like alerting. Machine learning insights can be extremely useful in heuristics and will play a larger role going forward.

Who Should Read This Book?

This book is aimed at helping software engineers feel at home and thrive in a modern application stack. As such, it provides just the right amount of introductory information about DevOps tasks. This is not to say it has nothing to offer established DevOps engineers. On the contrary, it provides plenty of useful information about containerization, monitoring, and troubleshooting. If you are a DevOps engineer or software engineer in a small shop, you can even use this book to help you create your whole application stack, from local development to production.

So, if you're a software developer looking for knowledge about DevOps, this book is for you. If you're interested in becoming more of a generalist, this book is for you. And if I've paid you money to read this book—well, this book is definitely for you.

How This Book Is Organized

This book is divided into three parts, as follows:

Part I: Infrastructure as Code, Configuration Management, Security, and Administration

Part I introduces the concepts of infrastructure as code (IaC) and configuration management (CM), which are essential for building systems with a repeatable, versioned, and predictable state. We'll also explore host-based and user-based security.

Chapter 1: Setting Up a Virtual Machine This chapter discusses the concepts of IaC and CM. It then introduces two technologies, Vagrant and Ansible, that you'll use to create and provision an Ubuntu VM.

Chapter 2: Using Ansible to Manage Passwords, Users, and Groups This chapter looks at how to use CM for user and group creation to restrict file and directory access. It also explains how to use CM to enforce complex passwords.

Chapter 3: Using Ansible to Configure SSH This chapter shows you how to set up public key and two-factor authentication over SSH, thus making it harder for unauthorized users to gain access to your host and sensitive data.

Chapter 4: Controlling User Commands with sudo This chapter shows you how to create a security policy that delegates command access for a specific user and group. Controlling the command access that users and groups have on a host can help you avoid unnecessary exposure to attackers. At a minimum, it prevents you from having a poorly configured OS.

Chapter 5: Automating and Testing a Host-Based Firewall This chapter describes how to create and test a minimal firewall that will block all unwanted access while permitting approved traffic. By limiting port exposure, you can reduce the vulnerabilities your host and application may encounter from the outside.

Part II: Containerization and Deploying Modern Applications

Part II introduces the concepts of containerization, orchestration, and delivery. It also explores some of the components that make up a modern stack.

Chapter 6: Containerizing an Application with Docker This chapter introduces containers and containerization, and it shows how to create a sample containerized application. Having a basic understanding of containers and how to use them for local development and production is key to your ability to work with any modern application stack.

Chapter 7: Orchestrating with Kubernetes This chapter introduces container orchestration and explores how to use technologies like Kubernetes and minikube to deploy an application on a local cluster. It also serves as an example of how to set up a local development environment.

Chapter 8: Deploying Code This chapter discusses the concept of continuous integration and continuous deployment (CI/CD). It also explores some core technologies, like Skaffold, that allow you to create a pipeline on a local Kubernetes cluster. After building an effective CI/CD pipeline, you'll have a good understanding of how to build, test, and deploy software.

Part III: Observability and Troubleshooting

Finally, Part III introduces the concepts of monitoring, alerting, and troubleshooting. It looks at metric collection and visualization for applications and hosts. It also discusses some common host and application issues, as well as tools you can use to diagnose them.

Chapter 9: Observability This chapter introduces the concept of a monitoring and alerting stack, and it explores the technologies (Prometheus, Alertmanager, and Grafana) that make up this stack. You'll learn how to detect a system's state and alert on it when things are out of scope.

Chapter 10: Troubleshooting Hosts The last chapter discusses common issues and errors on a host and some tools you can use to troubleshoot them. Being able to analyze issues on a host will help you in times of crisis and help you understand performance issues in your own code and applications.

What You'll Need

In order to explore the DevOps concepts in this book, you'll install some tooling and the free VirtualBox virtualization technology for x86 hardware that allows you to run other operating systems on your local host. Unfortunately, some of the tools needed for these tasks won't work natively on some OSes and CPUs, such as Windows and Apple Silicon. Using Linux

or an Intel-based Mac as the host machine is the most straightforward option. The following list summarizes what you can expect for each OS:

Linux

If you're on a Linux host, all the examples and sample applications will work out of the box. Since you'll be installing VirtualBox, you'll want to be running a desktop version of Linux rather than a headless server.

Intel-based Mac

If you're running an Intel-based Mac, as with Linux, all the examples and sample applications will work without any modifications. Use the Brew package manager (*https://brew.sh*) to install software.

Windows

If you're on a Windows host, installing all the tools and applications in this book can be a challenge. For example, you'll use Ansible to explore configuration management, but there's no easy way to install Ansible on Windows. As a workaround, you can use an Ubuntu VM as your starting point. I recommend creating the VM with Hyper-V, since it's native to Windows. You'll need Windows 10 or 11 Pro to use Hyper-V. See the Ubuntu Wiki (*https://wiki.ubuntu.com/Hyper-V*) for instructions on creating an Ubuntu VM on Hyper-V.

You'll also need to enable nested virtualization since you'll be installing VirtualBox inside the Hyper-V Ubuntu VM. To enable this feature, enter the following command in an administrative PowerShell terminal:

```
Set-VMProcessor -VMName VMName -ExposeVirtualizationExtensions $true
```

You'll need to run this command when the Ubuntu VM is stopped, or it will fail. Replace *VMName* with the name of the Ubuntu VM you just created.

After your VM is up and running, you'll install VirtualBox using the Ubuntu version listed at *https://www.virtualbox.org/wiki/Linux_Downloads*. After completing that installation, you'll be able to perform the book's examples from within the newly created VM.

For older versions of Windows, you can use VirtualBox (yes, VirtualBox within VirtualBox) or VMware (*https://www.vmware.com/products/workstation-player.html*) to create the Ubuntu VM. Instructions for these options are beyond the scope of this book.

Apple Silicon

If you're using an Apple Silicon computer as your host machine, VirtualBox is not an option. Apple Silicon's CPU is based off the ARM architecture, and VirtualBox works only on x86. Instead, you'll need

to use a virtualization technology like Parallels (*https://parallels.com*), VMware Fusion (*https://vmware.com*), or Qemu (*https://www.qemu.org*) to create an ARM-based virtual machine. The first two options are paid software and may provide a better user experience. Qemu is free and open source, and it requires some extra configuration steps. Visit the companion GitHub repository (*https://github.com/bradleyd/devops_for_the_desperate/tree/main/apple-silicon/*) for detailed instructions on how to set up a suitable lab to follow along on your Apple Silicon Mac.

To get the best experience, your host should have a minimum of 8GB of memory and at least 20GB of free disk space available; your mileage might vary if you have less. This book also makes some basic assumptions about your comfort level with Linux and the command line. You should be familiar with Bash and feel at home editing files.

Downloading and Installing VirtualBox

Download the installer from *https://www.virtualbox.org/wiki/Downloads/*. Choose the latest version and the correct download for your specific operating system. As mentioned previously, Windows users using Hyper-V will install VirtualBox for Ubuntu Linux. For Intel-based Macs, click the OS hosts link and download the installer. For Linux, you guessed it—click the Linux distributions link to find the download for your distribution. The VirtualBox website has excellent instructions for the different OSes at *https://www.virtualbox.org/manual/*.

Launch VirtualBox from where you installed it to verify that it works. If everything is okay, you should be greeted with a start screen (see Figure 1).

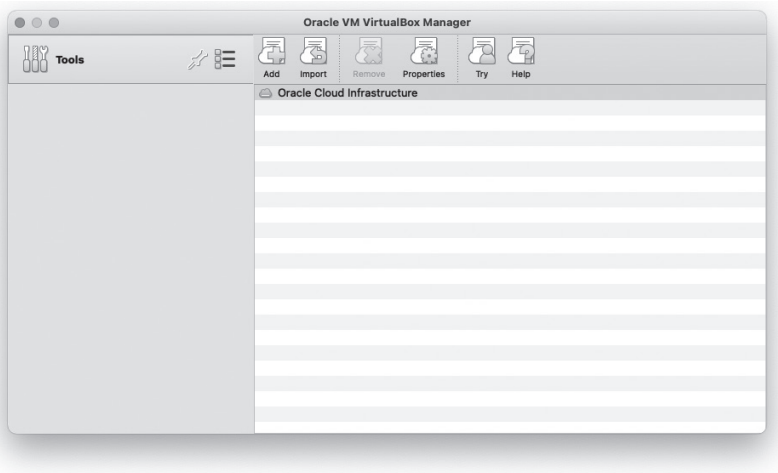


Figure 1: VirtualBox start screen on macOS (it will look different depending on your host OS)

If you decide to use your OS's package manager to install VirtualBox, make sure you've got the latest version, as older versions might show differences from the examples in this book.

WARNING *If you are running macOS, you'll need to allow VirtualBox extra permissions when trying to launch the virtual machine. You will be prompted to allow VirtualBox to control your computer, and you should go ahead and do so.*

Companion Repository

As this is a book for the desperate, I have taken the liberty of creating IaC files, Kubernetes manifests, an example application, and other things that will help you follow along throughout. I have put all the examples and source code in a Git repository located at *https://github.com/bradleyd/devops_for_the_desperate.git*. To follow along with the chapters and examples, you'll need to clone the book's repository. Your OS should have Git installed by default, but if it does not, visit *https://git-scm.com/downloads* for information on how to download and install Git for your specific OS.

From your terminal, enter the following command to clone the companion repository:

```
$ git clone https://github.com/bradleyd/devops_for_the_desperate.git
```

Feel free to clone this repository to anywhere you like. I have added some information in the *README* file as well if you need any additional guidance. We'll revisit this repository throughout this book.

Editor

Throughout this book, you'll need to edit and view files to complete tasks. For example, in some of the Ansible files, I've either left portions commented out that you'll need to uncomment, or you'll need to fill in some missing information.

I recommend using any editor you are comfortable with. You won't need any special plug-in or dependency to follow along in this book. However, if you look hard enough, I am sure you can find syntax plug-ins to help with editing the different types of files, like Ansible and Vagrant manifests. I use Vim as my editor, but feel free to substitute your favorite.

And now, with all the background out of the way, you are ready to get started! In Chapter 1, we'll dive into setting up a local virtual machine.

PART I

INFRASTRUCTURE AS CODE, CONFIGURATION MANAGEMENT, SECURITY, AND ADMINISTRATION

1

SETTING UP A VIRTUAL MACHINE

Provisioning (that is, setting up) a virtual machine (VM) is the act of configuring a VM for a specific purpose. Such a purpose could be running an application, testing software across a different platform, or applying updates.

Setting up a VM requires two steps: creating and then configuring it. For this example, you'll use Vagrant and Ansible to build and configure a VM. Vagrant automates the process of creating the VM, while Ansible configures the VM once it's running. You'll set up and test your VM locally, on VirtualBox. This process is similar to creating and provisioning servers in the cloud. The VM you set up now will be the foundation of all the examples in the first section of this book.

Why Use Code to Build Infrastructure?

Using code to build and provision infrastructure lets you consistently, quickly, and efficiently manage and deploy applications. This allows your

infrastructure and services to scale. It also can reduce operating costs, decrease time for recovery during a disaster, and minimize the chance of configuration errors.

NOTE *In the DevOps field, you'll often hear two terms that relate to creating and configuring infrastructure:* infrastructure as code (IaC) *and* configuration management (CM). *Treating infrastructure as code is the process of using code to describe and manage infrastructure like VMs, network switches, and cloud resources such as Amazon Relational Database Service (RDS). CM is the process of configuring those resources for a specific purpose in a predictable, repeatable manner. The two tools we are using, Vagrant and Ansible, are considered IaC and CM, respectively.*

Another benefit of treating your infrastructure as code is ease of deployment. Applications are built and tested the same way in a delivery pipeline. For example, artifacts like Docker images are created and deployed consistently, using the same versions of libraries and programs. Treating your infrastructure as code allows you to build reusable components, use test frameworks, and apply standard software engineering best practices.

There are times when treating your infrastructure as code may be overkill, however. For example, if you have only one VM to stand up or a simple Bash script to run, it may not be worth the time and effort to create all the infrastructure and CM code to accomplish something you can do in five minutes. Use your best judgment when deciding on the route to take.

Getting Started with Vagrant

Vagrant is a framework that makes it easy to create and manage VMs. It supports multiple operating systems (OSs) that can run on multiple platforms. Vagrant uses a single configuration file, called a *Vagrantfile*, to describe the virtual environment in code. You'll use this to create your local infrastructure.

Installation

To install Vagrant, visit Vagrant's website at *https://www.vagrantup.com/downloads.html*. Choose the correct OS and architecture for your host. To complete the installation, download the binary and follow the instructions specific to your OS. For example, since I am on a Mac, I would choose the macOS 64-bit link to download the latest version.

When your VM comes up, you'll also need to make sure that it has VirtualBox's guest additions installed on it. (You should have installed VirtualBox when following along with this book's Introduction.) *Guest additions* provide better driver support, port forwarding, and host-only networking. They help your VM run faster and have more options available.

After you have finished installing Vagrant, enter the following command in your terminal to install the Vagrant plug-in for guest additions:

```
$ vagrant plugin install vagrant-vbguest
Installing the 'vagrant-vbguest' plugin. This can take a few minutes...
Fetching vagrant-vbguest-0.30.0.gem
Installed the plugin 'vagrant-vbguest (0.30.0)'!
```

The output above shows a successful installation of the `vbguest` plug-in for Vagrant. Your version of the plug-in will most likely be different since new versions come out periodically. It is good practice to update this plug-in anytime you upgrade Vagrant and VirtualBox.

Anatomy of a Vagrantfile

A Vagrantfile describes how to build and provision a VM. It's best practice to use one Vagrantfile per project so you can add the configuration file to your project's version control and share it with your team. The configuration file's syntax is in the Ruby programming language, but you just need to understand a few basic principles to get started.

The Vagrantfile provided with this book contains documentation and sensible options to save you time. This file is too large to include here, so I'll discuss only the sections I changed from the Vagrant defaults. You'll start at the top of the file and work your way down to the bottom, so feel free to open it and follow along. It is located under the *vagrant/* directory in the repository you cloned from the Introduction. Later in this chapter, you'll use this file to create your VM.

Operating System

Vagrant supports many OS base images, called *boxes*, by default. You can search the list of boxes that Vagrant supports at *https://app.vagrantup.com/boxes/search/*. Once you find the one you want, set it near the top of the Vagrantfile using the `vm.box` option, as shown below:

```
config.vm.box = "ubuntu/focal64"
```

In this case, I've set the `vm.box` identifier to `ubuntu/focal64`.

Networking

You can configure the VM's network options for different network scenarios, like *static IP* or *Dynamic Host Configuration Protocol (DHCP)*. To do this, modify the `vm.network` option near the middle of the file:

```
config.vm.network "private_network", type: "dhcp"
```

For this example, you'll want the VM to obtain its IP address from a private network using DHCP. That way, it'll be easy to access resources like a web server on the VM from your local host.

Providers

A *provider* is a plug-in that knows how to create and manage a VM. Vagrant supports multiple providers to manage different types of machines. Each provider has common options like CPU, disk, and memory. Vagrant will use the provider's application programming interface (API) or command line options to create the VM. You can find a list of supported providers at *https://www.vagrantup.com/docs/providers/*. The provider is set near the bottom of the file and looks like this:

```
config.vm.provider "virtualbox" do |vb|
  vb.memory = "1024"
  vb.name = "dftd"
  --snip--
end
```

Basic Vagrant Commands

Now that you know how a Vagrantfile is laid out, let's look at some basic Vagrant commands. The four you'll use most often are `vagrant up`, `vagrant destroy`, `vagrant status`, and `vagrant ssh`:

`vagrant up` Creates a VM using the Vagrantfile as a guide

`vagrant destroy` Destroys the running VM

`vagrant status` Checks the running status of a VM

`vagrant ssh` Accesses the VM over Secure Shell

Each of these commands has additional options. To see what they are, enter a command and then add the `--help` flag for more information. To learn more about Vagrant's features, visit the documentation at *https://www.vagrantup.com/docs/*.

Once you create the VM by running **`vagrant up`**, you'll have a core Linux system with all the OS defaults. Next, let's look at how you can apply your own configuration to the system by provisioning it.

Getting Started with Ansible

Ansible is a CM tool that can orchestrate the provisioning of infrastructure like VMs. Ansible uses a *declarative configuration style*, which means it allows you to describe what the desired state of infrastructure should look like. This is different from an *imperative configuration style*, which requires you to supply all the minute details on your desired state of infrastructure. Because of its declarative style, Ansible is a great tool for software engineers who are not well versed in system administration. Ansible is also open-source software and free to use.

Ansible is written in Python, but you don't need to understand Python to use it. The one dependency you will need to understand is *Yet Another Markup Language (YAML)*, which is a data serialization language that

Ansible uses to describe complex data structures and tasks. It's easy to pick up simply by looking at some basic examples, and I'll provide a few when I review the Ansible playbook and tasks later. Two important things worth noting here are that YAML uses indentation to organize elements like Python, and it is also case sensitive. You can read more about YAML at *https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html*.

Ansible applies its configuration changes over *Secure Shell (SSH)*, which is a secure protocol to communicate with remote hosts. The most common use of SSH is to gain access to the command line on a remote host, but users can also deploy it to forward network traffic and copy files securely. By using SSH, Ansible can provision a single host or a group of hosts over the network.

Installation

Now, you should install Ansible so Vagrant can use it for provisioning. Head over to Ansible's documentation at *https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html*. Locate the documentation for your specific OS and follow the steps to install Ansible. For example, I am using macOS, and the preferred way to install Ansible on macOS is to use *pip*, which is a Python package manager used to install applications and dependencies. I found this information under the Installing Ansible on macOS link, which eventually directed me to install Ansible using pip under the Installing Ansible with pip link. Since Ansible is written in Python, using pip is an effective way to install the latest version.

Key Ansible Concepts

Now that you've installed Ansible, you'll need to know these terms and concepts to have it up and running quickly:

Playbook A *playbook* is a collection of ordered tasks or roles that you can use to configure hosts.

Control node A *control node* is any Unix machine that has Ansible installed on it. You will run your playbooks or commands from a control node, and you can have as many control nodes as you like.

Inventory An *inventory* is a file that contains a list of hosts or groups of hosts that Ansible can communicate with.

Module A *module* encapsulates the details of how to perform certain actions across operating systems, such as how to install a software package. Ansible comes preloaded with many modules.

Task A *task* is a command or action (such as installing software or adding a user) that is executed on the managed host.

Role A *role* is a group of tasks and variables that is organized in a standardized directory structure, defines a particular purpose for the server, and can be shared with other users for a common goal. A typical role could configure a host to be a database server. This role would include all the files and instructions necessary to install the database application, configure user permissions, and apply seed data.

Ansible Playbook

To configure the VM, you'll use the Ansible playbook I have provided. This file, named *site.yml*, is located in the *ansible/* directory you cloned from the Introduction. Think of the playbook as an instruction manual on how to assemble a host. Now, take a look at the playbook file itself. Navigate to the *ansible/* directory and open the *site.yml* file in your editor.

You can break up playbook files into different sections. The first section functions as the header, which is a good place to set global variables to use throughout the playbook. In the header, you'll set things like the `name` of the play, the `hosts`, the `remote_user`, and the privileged escalation method:

```
---
- name: Provision VM
  hosts: all
  become: yes
  become_method: sudo
  remote_user: ubuntu
--snip--
```

These settings are mostly boilerplate, but let's focus on a few points. Be sure to give each play a `name` so it's easier to find and debug if things go wrong. The `name` of the play in the example above is set to `Provision VM`. You could have multiple plays in a single playbook, but for this example, you'll need only one. Next, the `hosts` option is set to `all` to match any Vagrant-built VMs because Vagrant will autogenerate the Ansible inventory file dynamically. Some operations on a host will require elevated privileges, so Ansible allows you to *become*, or activate privilege escalation for, a specific user. Since you're using Ubuntu, the default user with escalated privileges is `ubuntu`. You also can set the different methods to use for authorization, and you'll use `sudo` for this example.

The next section is where you'll list all the tasks for the host. This is where the actual work is being done. If you think of the playbook as an instruction manual, the *tasks* are just the separate steps in that manual. The `tasks` section looks like this:

```
--snip--
  tasks:
   #- import_tasks: chapter2/pam_pwquality.yml
   #- import_tasks: chapter2/user_and_group.yml
--snip--
```

The built-in Ansible `import_tasks` function is loading tasks from two separate files: *pam_pwquality.yml* and *user_and_group.yml*. The `import_tasks` function allows you to organize the tasks better and avoid a large, cluttered playbook. Each of these files can have one or many individual tasks. I'll discuss tasks and other parts of the playbook in future chapters. For now, note that these tasks are commented out with the hash mark (`#`) symbol and will not change anything until you uncomment them.

Basic Ansible Commands

The Ansible application comes with multiple commands, but you'll mostly use these two: `ansible` and `ansible-playbook`.

You'll primarily use the `ansible` command for running ad hoc or one-time commands that you execute from the command line. For example, to instruct a group of web servers to restart Nginx, you would enter the following command:

```
$ ansible webservers -m service -a "name=nginx state=restarted" --become
```

This instructs Ansible to restart Nginx on a group of hosts called *webservers*. Note that the mapping for the *webservers* group would be located in the inventory file. The Ansible `service` module interacts with the OS to perform the restart. The `service` module requires some extra arguments, and they are passed with the `-a` flag. In this case, both the name of the `service` (`nginx`) and the fact that it should restart are indicated. You need *root* privileges to restart a `service`, so you'll use the `--become` flag to ask for privilege escalation.

The `ansible-playbook` command runs playbooks. In fact, this is the command Vagrant will use during the provisioning phase. To instruct `ansible-playbook` to execute the *aws-cloudwatch.yml* playbook against a group of hosts called *dockerhosts*, you would enter the following command in your terminal:

```
$ ansible-playbook -l dockerhosts aws-cloudwatch.yml
```

The `dockerhosts` need to be listed in the inventory file for the command to succeed. Note that if you do not provide a subset of hosts with the `-l` flag, Ansible will assume you want to run the playbook on all the hosts found in your inventory file.

Creating an Ubuntu VM

Up to this point, we've been discussing concepts and configuration files. Now, let's put that knowledge to use and stand up and provision some infrastructure. To create the Ubuntu VM, make sure you are in the same directory as the Vagrantfile. This is because Vagrant needs to reference the configuration file while creating the VM. You'll use the `vagrant up` command to create the VM, but before running the command, you should know that it produces a lot of output and may take a few minutes. Therefore, I'm focusing on only the relevant parts here. Enter the following command in your terminal:

```
$ vagrant up
```

The first section of the output to look at is Vagrant downloading the base image:

```
--snip--
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/focal64'...
--snip--
```

Here, Vagrant is downloading the `ubuntu` image, as expected. The image download may take a few minutes, depending on your internet connection.

Next, Vagrant will configure a public/private key pair to provide SSH access to the VM. (We'll discuss key pairs in more detail in Chapter 3.)

```
--snip--
default: Vagrant insecure key detected. Vagrant will automatically replace
default: this with a newly generated keypair for better security.
default:
default: Inserting generated public key within guest...
default: Removing insecure key from the guest if it's present...
default: Key inserted! Disconnecting and reconnecting using new SSH key...
--snip--
```

Vagrant stores the private key locally on your host (*.vagrant/*) and then adds the public key to the *~/.ssh/authorized_keys* file on the VM. Without these keys, you would not be able to connect to the VM over SSH.

By default, Vagrant and VirtualBox will mount a shared directory inside the VM. This shared directory will give you access to a host directory from within the VM:

```
--snip--
==> default: Mounting shared folders...
   default: /vagrant => /Users/bradleyd/devops_for_the_desperate/vagrant
--snip--
```

You can see that my local host directory *Users/bradleyd/devops_for_the_desperate/* is mounted at the *vagrant/* directory inside the VM. Your directory will be different. You can use this shared directory for transferring files like source code between host and VM. If you don't need the shared directory, Vagrant provides a configuration option to turn it off. See Vagrant's documentation for further details.

Finally, the following shows the Ansible `provisioner` output:

```
--snip--
==> default: Running provisioner: ansible...
    default: Running ❶ansible-playbook...

  PLAY [Provision VM] ******************************
❷ TASK [Gathering Facts] ***************************
❸ ok:  [default]

  PLAY RECAP ******************************
```

```
--snip--
default     : ok=1     ❹changed=0     unreachable=0     failed=0     skipped=0
rescued=0     ignored=0
```

This shows that the Ansible `provisioner` is run using the `ansible-playbook` ❶ command. Ansible logs each `TASK` ❷ and whether anything was changed on the host ❸. In this case, all the `tasks` are commented out, so nothing was `changed` ❹ on the VM. This output is the first place to look when gauging success or failure.

Let's perform a sanity check and see whether the VM is actually running. Enter the following command in your terminal to show the VM's current status:

```
$ vagrant status
Current machine states:
default     running (virtualbox)
--snip--
```

Here, you can see that the status of the VM is `running`. This means you created the VM, and it should be accessible over SSH.

If your output looks different, make sure there are no errors from the `vagrant up` command before continuing. If you need more information, add the `debug` flag to the `up` command to make Vagrant increase the output verbosity: **`vagrant up --debug`**. You'll need to have a successful provision at this point, or it will be difficult to follow along with the remaining chapters.

Summary

In this chapter, you installed Vagrant and Ansible to create and configure a VM. You learned how to configure Vagrant using its Vagrantfile, and you gained basic knowledge of how to provision a VM using Ansible playbooks and tasks. Now that you understand these basic concepts, you should be able to create and provision any type of infrastructure, not just VMs.

In the next chapter, you'll use two provided Ansible tasks to create a user and group. You'll need to have a foundation in user and group management when configuring a host.

2

USING ANSIBLE TO MANAGE PASSWORDS, USERS, AND GROUPS

Now that you've built your VM, let's move on to performing administrative tasks like user management. The DevOps practice of automation is key to building and managing resources. To manage any Linux host, you need a basic understanding of the workings of passwords, users, and groups. Users and passwords are the building blocks of identity management, while groups allow you to manage a collection of users and control access to files, directories, and commands. Dividing up responsibilities between users and groups can be the difference between allowing unauthorized access and thwarting it.

In this chapter, you'll continue learning how to use Ansible, and you'll also provision the VM you just created to improve your basic security policy. You'll use some provided Ansible tasks to enforce complex passwords, manage users and groups, and control access to a shared directory and file. Once you have learned those security basics, you'll be able to use them as the foundation of every playbook.

Enforcing Complex Passwords

Letting users decide what a strong password is can be a recipe for disaster, so you'll need to enforce complex passwords on every host that users can access. Since automation is one of our guiding principles, you'll use code to enforce strong passwords for all users. To do this, you can use an Ansible task to install a plug-in for *Pluggable Authentication Modules (PAM),* which is a user authentication framework that most Linux distributions employ. The plug-in to provide complex passwords is called `pam_pwquality`. This module validates passwords based on criteria you set.

Installing libpam-pwquality

The `pwquality` PAM module is available in the Ubuntu software repository under the name `libpam-pwquality`. You'll use the Ansible tasks provided with this book to install and configure this package. Remember, the goal is to automate everything you can, and tasks provide the mechanism to carry out administrative work. These tasks are located in the repository you cloned from the Introduction. Navigate to the *ansible/chapter2/* directory and open the *pam_pwquality.yml* file in your favorite editor. This file contains two tasks: `Install libpam-pwquality` and `Configure pam_pwquality`.

Let's focus on the first task that uses the Ansible `package` module to install `libpam-pwquality` on the VM. At the top of the file, the install task should look like this:

```
---
- name: Install libpam-pwquality
  package:
    name: "libpam-pwquality"
    state: present
--snip--
```

Each Ansible task should start with a `name` declaration that defines its goal. In this case, the `name` is `Install libpam-pwquality`. Next, the Ansible `package` module performs the software installation. The `package` module requires you to set two parameters: `name` and `state`. In this example, the package name (found in the Ubuntu repository) should be `libpam-pwquality`, and the `state` should be `present`. To remove a package, set the `state` to **`absent`**. This is a good example of declarative instruction, since you are telling Ansible to make sure this package is installed. You don't need to worry how it gets installed, as long as it does. If you install the package (`present`) and then delete the task from Ansible, the package will still be installed on the

next provision. You would have to explicitly set the package to `absent` if you wanted the host to represent your desired state.

As mentioned in Chapter 1, Ansible modules (like the one above) perform common actions on an OS, such as enabling a firewall, managing users, or (in this case) installing software. Ansible allows your actions to be *idempotent*, which means you can do a specific action over and over again and the result will be the same as it was the last time you executed the action. Because of this, you should automate all you can! You'll save time and avoid mistakes caused by manual fatigue. Imagine if you had to configure 1,000 machines a day. It would be almost impossible without automation!

Configuring pam_pwquality to Enforce a Stricter Password Policy

On a default Ubuntu system, password complexity is not as strong as it could be. It requires a minimum password length of six characters and executes only some basic complexity checks. To enforce more complexity, you'll want to configure `pam_pwquality` to set a stricter password policy.

A file named */etc/pam.d/common-password* handles configuration of the `pam_pwquality` module. This file is where the Ansible task makes the necessary changes to validate passwords. All you need to do is change one line in that file. A common way to edit a line using Ansible is with the `lineinfile` module, which allows you to change a line in a file or check whether a line exists.

With the `pam_pwquality` task file still open, let's review the second task from the top. It should look like this:

```
--snip--
- name: Configure pam_pwquality
  lineinfile:
    path: "/etc/pam.d/common-password"
    regexp: "pam_pwquality.so"

    line: "password required pam_pwquality.so minlen=12 lcredit=-1 ucredit=-1
           dcredit=-1 ocredit=-1 retry=3 enforce_for_root"
    state: present
--snip--
```

Once again, the task starts with a name, `Configure pam_pwquality`, that describes its intent. Then it tells Ansible to use the `lineinfile` module to edit the PAM password file. The `lineinfile` module requires the `path` of the file to which you want to make changes. In this case, it is the PAM password file */etc/pam.d/common-password*. Use a regular expression, or *regexp*, to find the line in the file you want to change. The regular expression locates the line that has `pam_pwquality.so` in it and replaces it with a new line. The replacement `line` parameter contains the `pwquality` configuration changes, which enforce more complexity. The options provided above enforce the following:

- A minimum password length of 12 characters
- One lowercase letter
- One uppercase letter

- One numeric character
- One nonalphanumeric character
- Three retries
- Disable root override

Adding these requirements will strengthen Ubuntu's default password policy. Any new passwords will need to meet or exceed these requirements, which will make brute-forcing user passwords a bit harder for attackers.

NOTE *The negative values in the configuration line above inform* `pam_pwquality` *that it must have at least "one of" for that category. See the* `pam_pwquality` *man page (enter* `man pam_pwquality`*) for further details.*

Close the *pam_pwquality.yml* file so you can move on to creating users with an Ansible module.

Linux User Types

When it comes to Linux, users come in three types: normal, system, and root. You can think of a *normal user* as a human account, and you'll create one of those next. Every normal user is typically associated with a password, a group, and a username. Think of a *system user* as a nonhuman account, such as the user Nginx runs as. In fact, a system user is almost identical to a normal user, but it is located in a different user ID (UID) range for compartmental reasons. A *root user* (or *superuser*) account has unrestricted access to the operating system. You can tell the root user by its UID, which is always zero. As with all your configurations, you'll use an Ansible module to do the heavy lifting when it comes to creating and configuring users.

Getting Started with the Ansible User Module

Ansible comes with the `user` module, which makes managing users very easy. It handles all the messy details for accounts, like shells, keys, groups, and home directories. You'll use the `user` module to create a new user called *bender*. Feel free to name it something else if you want, but since the examples in this book use the *bender* username going forward, don't forget to change the name in future chapters as well.

Open the *user_and_group.yml* file located in the *ansible/chapter2/* directory. This file contains the following five tasks:

1. Ensure group *developers* exists.
2. Create the user *bender*.
3. Assign *bender* to the *developers* group.
4. Create a directory named *engineering*.
5. Create a file in the engineering directory.

These tasks will create a group and a user, assign a user to a group, and create a shared directory and file.

Though it's counterintuitive, let's start by focusing on the second task on the list, which creates the user *bender*. (We'll get to the first task in the "Linux Groups" section on the next page.) It should look like this:

```
--snip--
- name: Create the user 'bender'
  user:
    name: bender
    shell: /bin/bash
    password: $6$...(truncated)
--snip--
```

This task, like all others, starts with a `name` that describes what it will do. In this case, it is `Create the user 'bender'`. You'll use the Ansible `user` module to create a user. The `user` module has many options, but only the `name` parameter is required. In this example, the `name` is set to `bender`. Setting a user's password at provision time can be useful, so set the optional `password` parameter field to a known password hash (more on this later). The `password` value, beginning with `$6`, is a cryptic hash that Linux supports. I have included a sample password hash for *bender* to show how you can automate this step. In the next section, I will walk through the process I used to generate it.

Generating a Complex Password

You can use many different methods to generate a password to match the complexity you set in `pam_pwquality`. As mentioned earlier, I've supplied a password hash for you that matches this threshold to save time. I used a combination of two command line applications, `pwgen` and `mkpasswd`, to create the complex password. The `pwgen` command can generate secure passwords, and the `mkpasswd` command can generate passwords using different hashing algorithms. The `pwgen` application is provided by the `pwgen` package, and the `mkpasswd` application is provided by a package named `whois`. Together, these tools can generate the hash that Ansible and Linux expect.

Linux stores password hashes in a file called *shadow*. On an Ubuntu system, the password hashing algorithm is SHA-512 by default. To create your own SHA-512 hash for Ansible's user module, use the commands below on an Ubuntu host:

```
$ sudo apt update
$ sudo apt install pwgen whois
$ pass=`pwgen --secure --capitalize --numerals --symbols 12 1`
$ echo $pass | mkpasswd --stdin --method=sha-512; echo $pass
```

Since these packages are not installed by default, you'll need to install them first with the APT package manager. The `pwgen` command generates a complex password that matches what you need to satisfy `pwquality` and saves it into a variable called `pass`. Next, the contents of the variable `pass` are piped into `mkpasswd` to be hashed using the `sha-512` algorithm. The final output

should contain two lines. The first line contains the SHA-512 hash, and the second line contains the new password. You can take the hash string and set the `password` value in the user creation task to change it. Feel free to try it!

WARNING *In a real production environment, you won't want to include a password hash in a version control system or inside an Ansible task, for obvious reasons. I included this example so you can have an easy way to create complex passwords that satisfy the `pam_pwquality` module. Use a tool like Ansible Vault to protect any sensitive information, like passwords or private keys. Ansible Vault stores these secrets in encrypted files instead of playbooks or tasks. Using this technique is beyond this book, but to learn more about Ansible Vault, visit* https://docs.ansible.com/ansible/latest/user_guide/vault.html.

Linux Groups

Linux groups allow you to manage multiple users on a host. Creating groups is also an efficient way to limit access to resources on a host. It is much easier to administer changes to a group than to hundreds of users individually. For the next example, I've provided an Ansible task to create a group called *developers* that you will use to limit access to a directory and a file.

Getting Started with the Ansible Group Module

Like the `user` module, Ansible has a `group` module that can manage creating and removing groups. Compared to other Ansible modules, the `group` module is very minimal; it can only create or delete a group.

Open the *user_and_group.yml* file in your editor to review the group creation task. The first task in the file should look like this:

```
- name: Ensure group 'developers' exists
  group:
    name: developers
    state: present
--snip--
```

The `name` of the task states that you want to make sure a group exists. Use the `group` module to create the group. This module requires you to set the `name` parameter, which is set to `developers` here. The `state` parameter is set to `present`, so it will create the group if it does not already exist.

The group creation task is the first one in the file, and that is not by accident. You need to create the *developers* group before executing any other tasks. Tasks are run in order, so you need to make sure the group exists first. If you tried to reference the group before creating it, you would get an error message stating that the *developers* group doesn't exist, and the provisioning would fail. Understanding Ansible's task order of operations is key to performing more complex operations.

Keep the *user_and_group.yml* file open as you continue reviewing the other tasks.

Assigning a User to the Group

To add a user to a group with Ansible, you'll leverage the `user` module once again. In the *user_and_group.yml* file, locate the task that assigns *bender* to the *developers* group (the third task from the top in the file). It should look like this:

```
--snip--
- name: Assign 'bender' to the 'developers' group
  user:
    name: bender
    groups: developers
    append: yes
--snip--
```

First is the `name` of the task, which describes its intention. The `user` module appends *bender* to the *developers* group. The `groups` option can accept multiple groups in a comma-separated string. By using the `append` option, you leave *bender*'s previous groups intact and add only the *developers*. If you omit the `append` option, *bender* will be removed from all groups except its primary group and the one(s) listed in the `groups` parameter.

Creating Protected Resources

With *bender*'s group affiliation sorted out, let's visit the last two tasks in the *user_and_group.yml* file, which deal with creating a directory (*/opt/engineering/*) and a file (*/opt/engineering/private.txt*) on the VM. You'll use this directory and file to test user access for *bender* later.

With the *user_and_group.yml* file still open, locate the two tasks. Start with the directory creation task (the fourth from the top in the file), which should look like this:

```
- name: Create a directory named 'engineering'
  file:
    path: /opt/engineering
    state: directory
    mode: 0750
    group: developers
```

First, as before, set the `name` to match the task's intent. Use the `file` module to manage the directory and its attributes. The `path` parameter is where you want to create the directory. In this case, it's set to */opt/engineering/*. Since you want to create a directory, set the `state` parameter to the type of resource you want to create, which is `directory` in this example. You can use other types here, and you'll see another one when you create the file later. The `mode`, or privilege, is set to `0750`. This number allows the owner (*root*) to read, write, and execute against this directory, while the group members are allowed only to read and execute. The execute permission is needed to enter the directory and list its contents. Linux uses octal numbers (`0750`, in this case) to define permissions on files and groups. See the `chmod` man page for more information on permission modes. Finally, set

the group ownership of the directory to the *developers* group. This means only the users in the *developers* group can read or list the contents of this directory.

The last task in the *user_and_group.yml* file creates an empty file inside the */opt/engineering/* directory you just created. The task, located at the bottom of the file, should look like this:

```
- name: Create a file in the engineering directory
  file:
    path: "/opt/engineering/private.txt"
    state: touch
    mode: 0770
    group: developers
```

Set the task name to what you want to do on the host. Use the file module again to create a file and set some attributes on it. The path, which is required, gives the file's location on the VM. This example shows creating a file named *private.txt* inside the */opt/engineering/* directory. The state parameter is set to touch, which means to create an empty file if it does not exist. If you need to create a nonempty file, you can use the copy or template Ansible modules. See the documentation for more details. The mode, or privileges, is set to read, write, and execute for any user in the group (0770). Finally, set the group ownership of the file to the *developers* group.

It is important to understand that there are many methods you can use to protect resources on a Linux host. Group restrictions are just a small piece of a larger authorization stack you would see in a production environment. I'll discuss different access controls in a later chapter. But for now, just know that with Ansible's tasks and modules, you can perform many common system configurations, such as securing files and directories across your whole environment.

Updating the VM

So far, we've been describing Ansible modules and reviewing the tasks that will provision the VM. The next step actually uses them. To provision the VM, you'll need to uncomment the tasks in the playbook under the *ansible/* directory. The *site.yml* file is the playbook you referenced in the provisioners section of your Vagrantfile.

Open the *site.yml* playbook file in your editor and locate the Chapter 2 tasks that look like this:

```
--snip--
tasks:
  #- import_tasks: chapter2/pam_pwquality.yml
  #- import_tasks: chapter2/user_and_group.yml
--snip--
```

They are commented out. Remove the hash marks (#) at the start of the two lines to uncomment them so Ansible can execute the tasks.

WARNING *Do not uncomment any other tasks, since that will cause unexpected consequences. You'll use the other tasks later in this book.*

The playbook should now look like this:

```
---
- name: Provision VM
  hosts: all
  become: yes
  become_method: sudo
  remote_user: ubuntu
  tasks:
    - import_tasks: chapter2/pam_pwquality.yml
    - import_tasks: chapter2/user_and_group.yml
--snip--
```

Both Chapter 2 tasks, `pam_pwquality` and `user_and_group`, are now uncommented, so they will execute the next time you provision the VM. Save and close the playbook file for now.

You created the VM in Chapter 1. If the VM is not running, however, enter the **`vagrant up`** command to start it again. With the VM running, all you need to do is issue the **`vagrant provision`** command from within the *vagrant/* directory to run the provisioner:

```
$ vagrant provision
--snip--
PLAY RECAP *********************************************************************
Default : ok=8    changed=7   unreachable=0   failed=0   skipped=0    rescued=0   ignored=0
```

The last line shows that the Ansible playbook ran and completed `8` actions. Think of *actions* as the tasks and other operations being run. Seven of those eight actions changed some state on the VM. The line shows that the provision is complete and had no errors or failed actions.

If your provision has failures, stop and try to troubleshoot them. Run the **`provision`** command again with the **`--debug`** flag, as shown in Chapter 1, to receive more information. You'll need a successful provision to follow along with the examples in this book.

Testing User and Group Permissions

To test the user and group permissions you just configured, you'll issue the **`ssh`** command for `vagrant` to access the VM. Make sure you are in the *vagrant/* directory so you have access to the Vagrantfile. Once there, enter the command below in your terminal to log in to the VM:

```
$ vagrant ssh
vagrant@dftd:~$
```

You should be logged in as the *vagrant* user, which is the default user Vagrant creates.

Next, to verify the user *bender* was created, you'll use the **getent** command to query the *passwd* database for the user. This command allows you to query entries in files like */etc/passwd*, */etc/shadow*, and */etc/group*. To check *bender*'s existence, enter the following command:

```
$ getent passwd bender
bender:x:1002:1003::/home/bender:/bin/bash
```

Your result should look similar to the output above. If the user was not created, the command will complete without any result.

Now, you should check whether the *developers* group exists and whether *bender* is a member of it. Query the *group* database for this information:

```
$ getent group developers
developers:x:1002:bender
```

The result should look like the output above, with a *developers* group and the user *bender* assigned to it. If the group did not exist, the command would have exited without any result.

For the final check, test that only members of the *developers* group can access the */opt/engineering/* directory and the *private.txt* file. To do this, try to access the directory and file once as the *vagrant* user and then again as the *bender* user.

While logged in as *vagrant*, enter the command below to list the */opt/engineering/* directory and its contents:

```
$ ls -al /opt/engineering/
ls: cannot open directory '/opt/engineering/': Permission denied
```

The output indicates that access is `denied` when trying to list files in */opt/engineering* as the *vagrant* user. This is because the *vagrant* user is not a member of the *developers* group and thus does not have read access to the `directory`.

Now, to test the file permissions for *vagrant*, use the **cat** command to view the */opt/engineering/private.txt* file:

```
$ cat /opt/engineering/private.txt
cat: /opt/engineering/private.txt: Permission denied
```

The same error occurs because the *vagrant* user does not have read permissions on the file.

The next test is to verify that *bender* has access to this same directory and file. To do this, you must be logged in as the *bender* user. Switch users from *vagrant* to *bender* using the **sudo su** command. (I'll cover the `sudo` command in Chapter 4.)

In your terminal, enter the following command to switch users:

```
vagrant@dftd:~$ sudo su - bender
bender@dftd:~$
```

Once you have successfully switched users, try the command to list the directory again:

```
$ ls -al /opt/engineering/
total 8
drwxr-x--- 2 root developers    4096 Jul  3 03:59 .
drwxr-xr-x 3 root root          4096 Jul  3 03:59 ..
-rwxrwx--- 1 root developers       0 Jul  3 04:02 private.txt
```

Now, as you can see, you have successfully accessed the directory and its contents as *bender*, and the *private.txt* file is viewable.

Next, enter the following command to check whether *bender* can read the contents of the */opt/engineering/private.txt* file:

```
$ cat /opt/engineering/private.txt
```

You use the **`cat`** command again to view the contents of the file. Since the file is empty, there is no output. More importantly, there are no errors from *bender*'s attempt to access the file.

Summary

In this chapter, you provisioned the VM using the following Ansible modules: `package`, `lineinfile`, `user`, `group`, and `file`. These modules configured the host to enforce complex passwords, manage a user and group, and secure access to a file and directory. These are common tasks a DevOps engineer would do in a typical environment. Not only did you expand your Ansible knowledge, but you learned how to automate basic security hygiene on the VM. In the next chapter, you'll continue with the provided tasks and increase SSH security to limit access to the VM.

3

USING ANSIBLE TO CONFIGURE SSH

SSH is a protocol and tool that provides command line access to a remote host from your own machine. If you are managing a remote host or a fleet of remote hosts, the most common way to access them is over SSH. Most servers are likely to be headless, so the easiest way to access them is from a terminal. Since SSH opens access to a host, misconfiguration or default installations can lead to unauthorized access. As with a lot of Linux services out of the box, the default security settings are adequate for most cases, but you will want to know how to increase security and then automate it. As an engineer, you should understand the steps required to lock down SSH on a host or hosts.

In this chapter, you'll learn how to use Ansible to secure SSH access to your VM. You'll do this by disabling password access over SSH, requiring public key authentication over SSH, and enabling two-factor authentication (2FA) over SSH for your user *bender*. You'll use a combination of some familiar Ansible modules, and you'll be introduced to some new ones. By the end of this chapter, you'll have a better understanding of how to enforce strict access to SSH and the automation steps required to do so.

Understanding and Activating Public Key Authentication

Most Linux distributions use passwords to authenticate over SSH by default. Although this is okay for many setups, you should beef up security by adding another option: *public key authentication*. This method uses a key pair, consisting of a public key file and a private key file, to confirm your identity. Public key authentication is considered best practice for authenticating users over SSH because potential attackers who want to hijack a user's identity need both a copy of a user's private key and the passphrase to unlock it.

When you create an SSH session with a key, the remote host encrypts a *challenge* with your public key and sends the challenge back to you. Because you are in possession of the private key, you can decode the message and send back a response to the remote server. If the server can validate the response, it will know you are in possession of the private key and will thus confirm your identity. To learn more about the key exchange and SSH, visit *https://www.ssh.com/academy/ssh/*.

Generating a Public Key Pair

To generate a key pair, you'll use the `ssh-keygen` command line tool. This tool, usually installed on Unix hosts by default as part of the `ssh` package, generates and manages authentication key pairs for SSH. There's a good chance you already have a public key pair on your local host, but for this book, let's create a new key pair so you don't interfere with it. You'll also add a passphrase to the private key. A *passphrase* is like a password, but it's usually longer (more like a group of unrelated words than a complex stream of characters). You add it so that if your private key ever fell into the wrong hands, the bad actors would need to have your passphrase to unlock it and spoof your identity.

NOTE *The following instructions are for Linux and macOS only.*

In a terminal on your local host, enter the following command to generate a new key pair:

```
$ ssh-keygen -t rsa -f ~/.ssh/dftd -C dftd
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase): <passphrase>
```

```
Enter same passphrase again: <passphrase>
Your identification has been saved in /Users/bradleyd/.ssh/dftd.
Your public key has been saved in /Users/bradleyd/.ssh/dftd.pub.
```

You first instruct `ssh-keygen` to create an `rsa` key pair that has a name of `dftd` (DevOps for the Desperate). If you do not specify a name, it defaults to *id_rsa*, which might override your existing local key. The `-C` flag adds a human-readable comment to the end of the key that can help identify what the key is for. Here, it's also set to `dftd`. During execution, the command should prompt you to secure your key by adding a passphrase. Enter a strong passphrase to protect the key. Also remember to always keep your passphrase safe, because if you lose it, your key will become forever locked and you will never be able to use it for authentication again.

After you confirm the passphrase, the private key and public key files are created under your local *~./ssh/* directory.

Using Ansible to Get Your Public Key on the VM

Each user's home folder on the VM has a file called *authorized_keys*. This file contains a list of public keys the SSH server can use to authenticate that user. You'll use this file to authenticate *bender* when accessing the VM over SSH. To do this, you need to copy the local public key you just created in the previous section (*/Users/bradleyd/.ssh/dftd.pub,* in my case) and append the contents of that file to the */home/bender/.ssh/authorized_keys* file on the VM.

To copy the file's content, you'll use a provided Ansible task. This task and all the other tasks related to this chapter are located in the cloned repository under the *ansible/chapter3/* directory.

Open the *authorized_keys.yml* file in your favorite editor to review the Ansible task. The first thing you should notice is that this file has only one task. It should look like this:

```
- name: Set authorized key file from local user
  authorized_key:
    user: bender
    state: present
    key: "{{ lookup('file', lookup('env','HOME') + '/.ssh/dftd.pub') }}"
```

First, set the `name` of the task to identify its intent. Use the Ansible `authorized_key` module to copy your public key from the local host over to *bender* on the VM. The `authorized_key` module is quite simple and requires that you set only the `user` and `key` parameters. In this example, it copies the local public key you made earlier into `bender`'s */home/bender/.ssh/authorized_keys* file. Set the `state` to `present`, as you want to add the key and not remove it.

To get the contents of the local public key, you'll use Ansible's evaluation expansion operators ({{ }}) and a built-in Ansible function called `lookup`. The `lookup` function retrieves information from outside resources, based on the plug-in specified as its first argument. In this example, `lookup` uses the `file` plug-in to read the contents of the *~/.ssh/dftd.pub* public key file. The full path to this public key file is constructed with the

`lookup env` plug-in and string concatenation denoted by the + sign. The final result should look similar to this if you're on a Mac: */Users/bradleyd/.ssh/dftd.pub*. If you are on Linux, it should like similar to this: */home/bradleyd/.ssh/dftd.pub*. The file path will be different, depending on your OS and username.

Adding Two-Factor Authentication

Security is built in layers. The more layers you have, the harder it is for an intruder to gain access. The next layer of security to add is *two-factor authentication (2FA)*, which validates a user's identity by using credentials and something that the user has, like a phone or device. The main goal of 2FA is to make it harder for someone to spoof your identity if your password or key is compromised.

Two-factor authentication relies on your providing two out of these three things: *something you know*, *something you have*, and *something you are*. Here are some examples of each:

Something you know: password or pin

Something you have: phone or hardware authentication device, such as a YubiKey

Something you are: fingerprint or voice

For this example, you'll use a *time-based one-time password (TOTP)* to satisfy the "something you have" portion, along with your public key for access. You'll use the `Google Authenticator` package to configure your VM to use TOTP tokens for logging in. These TOTP tokens are usually generated from an application like `oathtool` (*https://www.nongnu.org/oath-toolkit/*) and are valid for only a short period of time. I have taken the liberty of creating 10 TOTP tokens that Ansible will use for you, but I will also show you how to use `oathtool` (more on this later).

To enforce 2FA on your VM, you'll use some provided Ansible tasks to install another PAM module, configure the SSH server, and enable 2FA. To review the provided tasks, first open the *two_factor.yml* file in your editor. (All the Ansible files for this chapter are located in the *ansible/chapter3/* directory.) This file has seven tasks, and each task has a specific job to enable 2FA. The tasks are named as follows:

1. Install the `libpam-google-authenticator` package.
2. Copy over preconfigured `GoogleAuthenticator` config.
3. Disable password authentication for SSH.
4. Configure PAM to use `GoogleAuthenticator` for SSH logins.
5. Set `ChallengeResponseAuthentication` to `Yes`.
6. Set Authentication Methods for *bender*, *vagrant*, and *ubuntu*.
7. Insert an additional line here that reads: Restart SSH Server.

We'll look at each of these tasks in the following sections.

Installing Google Authenticator

`Google Authenticator` is a PAM module that allows you to enforce 2FA over SSH. This module is located in the Ubuntu software repository under the name `libpam-google-authenticator`. The package contains all the necessary files to enable `Google Authenticator`. With the *two_factor.yml* file still open, find the first task at the top. It should look like this:

```
- name: Install the libpam-google-authenticator package
  apt:
    name: "libpam-google-authenticator"
    update_cache: yes
    state: present
```

The `name` on the first line identifies the task's intent (installing a package). You'll use Ansible's `apt` module to install the OS package. The `apt` module also requires the following `name` parameter to be set, and in this example, it is set to the package name `libpam-google-authenticator`.

NOTE *I chose to use the `apt` module because it's the default on Ubuntu, and it updates the OS's package manager cache before installing the `libpam-google-authenticator` package. The* package cache *is like a list of software titles that the OS knows about. If the package manager cache is stale, `apt` may not know how to find the package and the task may fail.*

Finally, as before, set the `state` to `present` since you want to install the package and not remove it. Most Ansible modules have the `state` set to `present` as a default, but you are most likely not the only person using these tasks. Letting the other engineers know your intent leaves little room for doubt or error, so even though you could omit this step, it's always better to be explicit.

Configuring Google Authenticator

To configure `Google Authenticator` for a user, you typically would run the `google-authenticator` command that was installed from the `libpam-google-authenticator` package. This application creates a configuration file named *.google_authenticator* in the user's *home/* directory by default. The configuration file consists of a Base32 key (secret); configuration options, such as token reuse and time to live; and 10 emergency recovery tokens. To keep the focus on provisioning, I've created the *google_authenticator* configuration file for you in the *chapter3/* directory.

WARNING *Do not use this file in the real world, as the secret key and tokens are not so secret. Instead, keep these values safe by storing them in Ansible Vault or another product like HashiCorp's vault (*https://www.vaultproject.io/*). You can also add the `no_log: True` option to any task that might write sensitive information to the provision log.*

Since the goal is to automate, you'll use an Ansible task to copy this configuration file over to the VM. If you're tempted to think, "It would be easier just to run the command by hand," remember that in most cases you'll

be managing many hosts. Doing that by hand would be tedious and make you error prone.

With the *two_factor.yml* file still open, locate the task on line 7 of the file that looks like this:

```
- name: Copy over preconfigured GoogleAuthenticator config
  copy:
    src: ../ansible/chapter3/google_authenticator
    dest: /home/bender/.google_authenticator
    owner: bender
    group: bender
    mode: 0600
```

As always, the `name` of the task describes its intent (copy a file). The Ansible `copy` module copies the configuration file from your local host to the VM. Use the `copy` module when you need to copy a file from a source to a destination. (The source can be either local or remote.) The `copy` module requires you to set the `src` and `dest` parameters. In this case, the `src` field is set to the local *google_authenticator* file in the cloned repository (*https://github.com/bradleyd/devops_for_the_desperate/*). Notice the two dots (`..`) in the beginning of the source (*src*) file. These dots indicate that the file is located one directory up from the current *vagrant/* directory, where the `ansible` command is run. Without these dots, the `ansible-playbook` command would not be able to find the *ansible/* directory where the file is located. The `dest` parameter is set to the file named */home/bender/.google_authenticator* on the VM. The file permission, or `mode`, is set to read and write (`0600`), so only the owner of the file, *bender*, can read and write to it.

To learn more about `Google Authenticator`, visit *https://github.com/google/google-authenticator/wiki/*.

Configuring PAM for Google Authenticator

As mentioned in Chapter 2, PAM controls a lot of authorization and authentication methods in Linux. To be able to use `Google Authenticator` over SSH, you need to modify the SSH PAM configuration file, which is very similar to what you did in Chapter 2. To add `Google Authenticator` to PAM, you'll need to make changes to the module file located at */etc/pam.d/sshd*. This file controls how PAM interacts with the SSH server (more on that later).

You'll use two provided Ansible tasks that disable password prompts over SSH and tell PAM where it can find the `Google Authenticator` file (*pam_google_authenticator.so*). Remember, you want to force users to use public key authentication in lieu of passwords. This change will also make it harder for attackers to brute-force SSH with a password since you will not allow it.

With the *two_factor.yml* file still open, locate the first of the two tasks that configure PAM (on line 15). It should look like this:

```
- name: Disable password authentication for SSH
  lineinfile:
    dest: "/etc/pam.d/sshd"
    regex: "@include common-auth"
    line: "#@include common-auth"
```

This task disables `password` prompts for SSH via the PAM module. To edit the PAM *sshd* file, this task uses the familiar Ansible `lineinfile` module, which locates the `common-auth` line with a regular expression (`regex`) and comments it out with a # sign. In this case, the regular expression searches for the full `common-auth` line. By commenting out that line, SSH `password` prompts for users are disabled when logging in over SSH.

The second task that will configure PAM, located on line 21, should look like this:

```
- name: Configure PAM to use GoogleAuthenticator for SSH logins
  lineinfile:
    dest: "/etc/pam.d/sshd"
    line: "auth required pam_google_authenticator.so nullok"
```

This task tells PAM about the `Google Authenticator` module. It uses the Ansible `lineinfile` module again to edit the PAM *sshd* file. This time, you just want to add the `auth` line to the bottom of the PAM file, which lets PAM know it should use `Google Authenticator` as an authentication mechanism. The `nullok` option at the end of the line tells PAM that this authentication method is optional, which allows you to avoid locking out users until they have successfully configured 2FA. In a production environment, you should remove the `nullok` option once all users have enabled 2FA.

Configuring the SSH Server

The SSH server manages all the SSH connections from the clients and enforces specific rules governing those connections. The SSH server will require some changes to expect a 2FA response, since that's not a default configuration.

First, you'll want to use Ansible to enable a keyboard response prompt when authenticating over SSH. The option to set is called `ChallengeResponseAuthentication`, and it's needed so users can enter the two-factor verification code when logging in.

The second change Ansible will make is to set the SSH users' `AuthenticationMethods`, which enable the SSH server to enforce specific ways for users to authenticate themselves. For this example, you'll set the `AuthenticationMethods` for *bender* to be `publickey` and `keyboard-interactive`. This will force *bender* to need a public key and a TOTP token to log in. You'll also set the *vagrant* and *ubuntu* users' `AuthenticationMethods` only to `publickey` to log in, so you'll still have users that can access the VM if anything goes wrong with 2FA.

With the *two_factor.yml* file still open, let's review the two tasks that modify the VM's SSH server. The first of these tasks, on line 26, should look like this:

```
- name: Set ChallengeResponseAuthentication to Yes
  lineinfile:
    dest: "/etc/ssh/sshd_config"
    regexp: "^ChallengeResponseAuthentication (yes|no)"
    line: "ChallengeResponseAuthentication yes"
    state: present
```

The task sets the `ChallengeResponseAuthentication` to `yes`. It uses the `lineinfile` module again to change a line in the VM's SSH server config file. It locates the line using a regular expression that searches for the `ChallengeResponseAuthentication` option at the beginning of a line that is set to `yes` or `no`. Once it finds the line, it sets the line to `ChallengeResponseAuthentication yes` to enable keyboard interactivity for 2FA.

The last task in the file that configures the SSH server should look like this:

```
- name: Set Authentication Methods for bender, vagrant, and ubuntu
  blockinfile:
    path: "/etc/ssh/sshd_config"
    block: |
      Match User "ubuntu,vagrant"
          AuthenticationMethods publickey
      Match User "bender,!vagrant,!ubuntu"
          AuthenticationMethods publickey,keyboard-interactive
    state: present
  notify: "Restart SSH Server"
```

This task sets the authentication methods for users using the `blockinfile` module. Similar to `lineinfile`, `blockinfile` can manipulate a block of text. This is useful when you need to change multiple lines at once and preserve indentation inside a file. The `blockinfile` module requires that the `path` parameter be set. In this case, the `path` of the file to edit is */etc/ssh/sshd_config*. The pipe character (`|`) is YAML notation for introducing a multiline string: the block of text, where the task uses an SSH server configuration option called `Match` that allows you to apply certain criteria to specific users. In this example, you want to allow the *ubuntu* and *vagrant* users to use `publickey` authentication only when logging in over SSH. Then you want to set the authentication methods for *bender* to be `publickey` and `keyboard-interactive`, to enforce 2FA. Finally, this example sets a `notify` action to `"Restart SSH Server"` on this task. (I'll discuss the `notify` option next.)

NOTE *The* sshd_config *file includes options to disable password prompts and PAM. You want to leave these options commented out, or not used, as you want to funnel all your authentication through PAM to keep with system defaults for accounting and sessions.*

Restarting the SSH Server with a Handler

Editing the configuration file is not enough; the SSH server requires a restart for all the changes to take effect. To make that happen, you'll use the `notify` Ansible option that triggers a `handler` to perform a single task. A `handler` is just like any other task, but it's executed only once and has a globally unique name across the whole playbook.

The last Ansible task in *two_factor.yml* activates a `handler` that restarts the SSH server for you. Open the *handlers/restart_ssh.yml* file found in the *ansible/* directory. It should look like this:

```
- name: Restart SSH Server
  service:
    name: sshd
    state: restarted
```

This `handler`'s `name` is set to `Restart SSH Server`. This `name` matches the `notify` value from the previous task (`Set Authentication Methods for bender, vagrant, and ubuntu`). This is not an accident. The values must match exactly to be triggered. The `service` module restarts the SSH server. This module requires the `name` parameter, which is `sshd` in this case, to be set. Finally, this task sets the `state` to `restarted`. If, for some reason, the SSH server does not restart, the task will fail.

You're now finished with the Ansible tasks, so it's safe to close all the open files.

Provisioning the VM

To provision the VM with all the tasks described thus far, you'll need to uncomment them in the playbook. You'll follow essentially the same process that you followed in Chapter 2, but this time around, you'll need to uncomment two tasks and a `handler`. Open the *site.yml* file in your editor and locate the task for authorized keys, which should look like this:

```
#- import_tasks: chapter3/authorized_keys.yml
```

Remove the `#` symbol to uncomment it.

Next, find the task for 2FA:

```
#- import_tasks: chapter3/two_factor.yml
```

Remove the `#` symbol to uncomment that line as well.

Next, find the `handler` section that's located below all the tasks. The `handler` to restart the SSH server should look like this:

```
#- import_tasks: handlers/restart_ssh.yml
```

Remove the `#` symbol at the beginning of the line to uncomment it.

The playbook should now look like this:

```
- name: Provision VM
  hosts: all
  become: yes
  become_method: sudo
  remote_user: ubuntu
  tasks:
    - import_tasks: chapter2/pam_pwquality.yml
```

```
    - import_tasks: chapter2/user_and_group.yml
    - import_tasks: chapter3/authorized_keys.yml
    - import_tasks: chapter3/two_factor.yml
  --snip--
  handlers:
    - import_tasks: handlers/restart_ssh.yml
```

Here, the changes to the playbook for Chapter 3 are added on to the changes from Chapter 2. As mentioned previously, the playbook is a collection of tasks that will perform specific actions on a host or group of hosts to enforce a specified state.

Now, you'll automate the configuration of the VM using Vagrant. Navigate to the *vagrant/* directory, and once there, enter the following command:

```
$ vagrant provision
--snip--
PLAY RECAP *********************************************************************
default    : ok=16   changed=9   unreachable=0   failed=0   skipped=0   rescued=0   ignored=0
```

Notice that the total task count has increased to `16` since the last provision. You have also changed a total of `9` things on the VM. Here's a summary of the things that changed:

- Seven new tasks from Chapter 3
- One task that updates the empty file from the previous chapter
- One `handler`

Once again, make sure no actions failed before you continue. The values from the provision output will vary, depending on how many times you run the `provision` command in this chapter. This is because Ansible is working hard to make sure your environment is consistent, and it doesn't do extra work that is not needed. As mentioned earlier, Ansible is idempotent, meaning it can be executed several times and each execution completes with the same end state you would expect from the initial execution.

Testing SSH Access

With the VM successfully provisioned, you should test *bender*'s access over SSH. To test public key and 2FA over SSH, you'll need the private key you created earlier and one of the emergency tokens from the *google_authenticator* file in the repository. The private key should be located in your local SSH directory. On my Mac, it's in */Users/bradleyd/.ssh/dftd*. The emergency tokens are the 10 eight-digit numbers located at the bottom of the *ansible/chapter3/google_authenticator* file. Choose the first one.

To `ssh` in to the VM as *bender*, open a terminal on your local host and enter the following command:

```
$ ssh ❶-i ~/.ssh/dftd -p 2222 ❷bender@localhost
Enter passphrase for key /Users/bradleyd/.ssh/dftd: <passphrase>
```

```
Verification code: <76338876>
--snip--
bender@dftd:~$
```

In the `ssh` command, you set the identity file to your private key ❶ for authentication and set the remote SSH port to `2222`. The default SSH port is 22, but Vagrant listens on a different SSH port to avoid conflicts on your local host. You also set the login user to *bender* and the SSH host to `localhost` ❷.

The output indicates you should have been prompted twice during this login session: once to enter the passphrase to unlock your private key, and a second time to enter a 2FA verification code. After satisfying both prompts, you should be successfully logged in to the VM as *bender*.

If, for some reason, you weren't prompted for a TOTP token or for the private key passphrase, stop and check for errors. You can log in to the VM as the *vagrant* user and inspect the logs. A good place to start looking for errors is in either */var/log/auth.log* or */var/log/syslog* on the VM. Common errors include the SSH server not restarting cleanly and one of the configuration files having a syntax issue.

Each of the 10 tokens provided is for one-time use. Every time you successfully use one, it's removed from the */home/bender/.google_authenticator* file. If, for some reason, you burn through all the tokens, run the **`vagrant provision`** command again to replace the file and replenish the tokens. Another option is to use a TOTP application like `oathtool` and generate a time-based one-time token by using the Base32 secret at the top of the */home/bender/.google_authenticator* file. You can install `oathtool` with Ubuntu's package manager by using the **`apt install oathtool`** command. Every time you need a token, you can use the following command:

```
$ oathtool --totp --base32 "QLIUWM4UVD7E5SI6PPVZ2EGRFU"
097903
```

Here, you pass `oathtool` your Base32 secret in the double quotes and set the flags `--totp` and `--base32` to generate the token. In this result, the token `097903` is generated and can be used when prompted for a verification code. Feel free to use this method or the provided tokens when logging in.

Summary

In this chapter, you secured the VM by disabling password logins, requiring public key authentication, and enforcing 2FA for *bender*. Automating these simple steps improves your host's security, whether it's local or on someone else's computer in the cloud. As with the previous chapters, these automation tasks are a part of a foundational base that you can employ with all your hosts. In the next chapter, you'll use more Ansible tasks to control user access by enabling security policies.

4

CONTROLLING USER COMMANDS WITH SUDO

So far, you have secured access to your VM with a public key and two-factor authentication. You have also controlled access to a specific file and directory, using group permissions. The next foundational piece is allowing users to run elevated commands on the VM. Users typically need access to commands that may require administrative permissions, such as restarting a service or installing a missing package. As an administrator, you want to keep a tight control on who can run which commands. On Linux operating systems, the `sudo` (superuser do) command allows users to run specific commands as *root* or another user while keeping an audit trail of events.

In this chapter, you'll use Ansible to install a simple Python Flask web application. You'll also use Ansible to create a *sudoers* security policy, which is configured by a file and determines what permissions users have when they invoke the `sudo` command. This policy will allow members of the *developers* group to use the `sudo` command to start, stop, restart, and edit the sample web application. Although this is a made-up example, it follows a typical release workflow that software engineers should be accustomed to. By the end of the chapter, you'll have a good grasp of how to automate application deployment and control it with a *sudoers* policy.

What Is sudo?

If you are new to `sudo`, it is a command line tool on most Unix OSs that allows a user or group of users to run commands as another user. For example, a software engineer may need to restart an Nginx web server that's owned by the *root* user, or a system administrator may need elevated permissions to install some software packages. If you have been around Linux long enough, you have likely used `sudo` to run a command that requires elevated privileges. Normally, you would not allow just anyone to have such privileges, due to various security implications. Regardless of your use case, users will need a safe and accountable way to access privileged commands to do their jobs.

One of the best features of `sudo` is its ability to leave an audit trail. If someone runs a command with `sudo`, you can check the log to see who ran what command. Without `sudo`, there is zero accountability if you blindly allow people to switch to other users to run commands.

You also can enhance `sudo` with plug-ins. In fact, `sudo` comes with a default security policy plug-in called *sudoers*, which determines what permissions users have when they invoke the `sudo` command. You'll implement this policy for your user *bender*.

Planning a sudoers Security Policy

When you are planning a *sudoers* policy, less is more. You want a user or group of users to have just the right amount of permissions on a host. If you have a user that can run many privileged commands while administering the company website at the same time, you'll have serious issues if that user is compromised. This is because any attackers will inherit the same access that the compromised user has.

That said, it is naive to think you can lock down a host completely and still get things done. Imagine a software delivery workflow where an application needs to be restarted after each deployment. Without proper user permissions, you will not be able to automate continuous delivery for that application.

For the example security policy you'll set up in this chapter, everyone in the *developers* group will be able to access the sample web application. They'll also be able to stop, start, and edit the main application file.

Installing the Greeting Web Application

The sample Python web application I have provided is cleverly (and lazily) named *Greeting*. This simple web application responds with an enthusiastic "Greetings!" when you visit *http://localhost:5000* on the VM. I am providing this application so you can focus on learning automation and provisioning; I won't go over its code here.

You'll use Ansible tasks to install the necessary libraries and files to run the web application. You'll also install a *systemd* unit file, the standard service manager that manages processes and services on a Linux host, to make it easier to start and stop the web application.

The Ansible tasks to install the web application (and all the other tasks for this chapter) are located in the *ansible/chapter4/* directory. You should navigate to that directory and open the task file named *web_application.yml* in your favorite editor.

This file contains four individual tasks, named as follows:

1. Install `python3-flask`, `gunicorn3`, and `nginx`
2. Copy Flask Sample Application
3. Copy *Systemd* Unit file for Greeting
4. Start and enable Greeting Application

I'll go over each of these tasks, starting with the one that installs the web application dependencies: `python3-flask`, `gunicorn3`, and `nginx`. It's the first task at the top of the file, and it should look like this:

```
- name: Install python3-flask, gunicorn3, and nginx
  apt:
    name:
      - python3-flask
      - gunicorn3
      - nginx
    update_cache: yes
```

The task `name` describes its intent, which is to `Install` some software packages. The `apt` module is used again to install the `python3-flask`, the `gunicorn3`, and the `nginx` packages from the Ubuntu repository on the VM. This time, however, the `apt` module uses some syntactical sugar: a YAML list. This feature allows you to install multiple packages (or remove them) in a single task, instead of having to create a task for each package you want to install.

NOTE *Flask (*https://palletsprojects.com/p/flask/*) is a web framework that is written in Python and known for its small code base and easy-to-use syntax. Gunicorn (*https://gunicorn.org/*), or Green Unicorn, is an HTTP server that is built on top of the web server gateway interface (WSGI,* https://wsgi.readthedocs.io/en/latest/*) standard. Gunicorn sits in front of the Flask application and proxies requests.*

The second task from the top copies the sample Greeting application over to the VM. You need two files to bring the Greeting web application to life, and the task should look like this:

```
- name: Copy Flask Sample Application
  copy:
    src: "../ansible/chapter4/{{ item }}"
    dest: "/opt/engineering/{{ item }}"
  group: developers
  mode: '0750'
  loop:
    - greeting.py
    - wsgi.py
```

The `copy` module copies the two files from the provided repository to the VM. The `src` and `dest` lines are templated (with double curly brackets) and replaced by the values from the `loop` module. Here, the `loop` module references two files by name: *greeting.py* and *wsgi.py*. The *greeting.py* file is the actual Python Flask code, while the *wsgi.py* file contains the application object for the HTTP server. During this task's runtime, the placeholder `{{ item }}` will be replaced with each of these two filenames from the `loop`. For example, the `src` line will look like `"../ansible/chapter4/greeting.py"` after the first pass of the `loop`. The `mode` line sets the permissions on both files to be read and to execute for anyone in the *developers* group.

Next, let's look at the task that copies the *systemd* unit file over to the VM. This task, located third from the top, should look like this:

```
- name: Copy Systemd Unit file for Greeting
  copy:
    src: "../ansible/chapter4/greeting.service"
    dest: "/etc/systemd/system/greeting.service"
```

This task starts with a descriptive `name`, as usual. Then, the familiar Ansible `copy` module copies a file from the local host to the VM. In this case, it copies the *greeting.service* file to a place on the VM where `systemd` can find it: */etc/systemd/system*.

Let's review the *system service* file. Such files can have many options and settings, but for this example, I've provided a simple one to control the Greeting web application's life cycle.

Open the *ansible/chapter4/greeting.service* file in your editor. It should look like this:

```
[Unit]
Description=The Highly Complicated Greeting Application
After=network.target

[Service]
Group=developers
WorkingDirectory=/opt/engineering
ExecStart=/usr/bin/gunicorn3 --bind 0.0.0.0:5000 --access-logfile - --error-logfile - wsgi:app
ExecReload=/bin/kill -s HUP $MAINPID
```

```
KillMode=mixed

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

The `WorkingDirectory` and `ExecStart` lines are the most important in this file. The first sets the working directory to */opt/engineering*, since that's where your application code lives. In the `ExecStart` line, the `gunicorn3` application calls the *wsgi.py* file to start the web application. You'll also tell `gunicorn3` to log STDOUT (`--access-logfile -`) and STDERR (`--error-logfile -`) to the *systemd* journal, which is forwarded by default to the */var/log/syslog* file. Close the *greeting.service* file for now.

The last task in the *web_application.yml* file ensures that the Greeting web application is started and that the `systemd` daemon is reloaded each time a provision is run. It should look like this:

```
- name: Start and enable Greeting Application
  systemd:
    name: greeting.service
    daemon_reload: yes
    state: started
    enabled: yes
```

Here, the `systemd` Ansible module starts the Greeting web application. The module requires you to set the `name` and `state`, which in this case are `greeting.service` and `started`, respectively. The `enabled` parameter tells `systemd` to start the service automatically during startup. Using the `daemon_reload` parameter also forces `systemd` to reload all service files and discover the *greeting.service* file before doing anything else. It's equivalent to running `systemctl daemon-reload`. The `daemon_reload` parameter is useful on the first provision of a host to make sure `systemd` knows about the service. Be sure to use the `daemon_reload` parameter so that `systemd` always knows about any changes to the service file.

NOTE *In Chapter 8, you'll see more advanced examples to help you learn how to deploy an application using a CI/CD pipeline inside Kubernetes.*

Anatomy of a sudoers File

A *sudoers* file is the place where you configure security policies (for users and groups) that invoke the `sudo` command. This type of security file is composed of sections called `Defaults`, `User Specifications`, and `Aliases`. A *sudoers* file is read from the top down, and since rules are applied in that order, the last matching rule always wins.

The `Defaults` syntax allows you to override some *sudoers* options at runtime, such as setting environment variables that users have access to when they run `sudo`. The `User Specifications` section determines which commands users can run and on which host they can run them. For example, you could give the *bender* user permission to run the `apt install` command on all web

server hosts. The `Aliases` syntax references other objects inside the file, and that is useful for keeping the configuration clear and concise when there is a lot of duplication.

The four aliases you can mix and match are as follows:

`Host_Alias` A host or a group of hosts

`Runas_Alias` A list of users or groups a command can be run as

`Cmnd_Alias` Specifies a command or multiple commands

`User_Alias` A user or group of users

For this example, you'll only use `Cmnd_Alias` and `Host_Alias` in your *sudoers* file.

Creating the sudoers File

To create the *sudoers* file, you'll use the Ansible `template` module and a template file. The Ansible `template` module is useful for creating files that will require some modification with variables. The `template` module creates files using the Jinja2 template engine for Python templates. You'll keep template files in a separate directory called *ansible/templates/* (more on this later).

NOTE *Jinja2 is a modern templating engine for the Python language. It is modeled after the Django web application templates.*

In the *ansible/chapter4/* directory, open the task file named *sudoers.yml* in your favorite editor. The first thing you should notice, at the top of the file, is a new Ansible module called `set_fact`. This module allows you to set host variables that can be used in a task or across a playbook. Here, you'll set a variable with it for use in your template file:

```
- set_fact:
    greeting_application_file: "/opt/engineering/greeting.py"
```

This creates a variable named `greeting_application_file` and sets its value to */opt/engineering/greeting.py* (where the previous tasks will install the web application). As noted previously, anyone in the *developers* group can read and execute in the */opt/engineering/* directory.

Next, locate the task right below the `set_fact` module. This task creates the *sudoers* file for the *developers* group and should look like this:

```
- name: Create sudoers file for the developers group
  template:
    src: "../ansible/templates/developers.j2"
    dest: "/etc/sudoers.d/developers"
    validate: 'visudo -cf %s'
    owner: root
    group: root
    mode: 0440
```

The Ansible `template` module builds out your *sudoers* file. It requires a source file (`src`) and a destination file (`dest`). The source file is your local Jinja2 template (*developers.j2*), and the destination file will be the *developers sudoers* file on the VM. The `template` module also contains a `validate` step to verify whether the template is correct. In this case, the `visudo` command edits and validates your *sudoers* file in a safe manner. Adding the `-cf` flag to `visudo` makes sure the *sudoers* file is compliant and free of syntax errors. The `%s` is a placeholder for the file in the `dest` parameter. If the `validate` command fails for any reason, the Ansible task will fail, too. Finally, set the owner, group, and permissions of the file to `root`, `root`, and `0440` (respectively). This is what *sudoers* is expecting for proper permissions.

The sudoers Template

The Ansible `template` module task referenced a source Jinja2 template file located in the *ansible/templates/* directory. It has the building blocks of your *sudoers* policy for the *developers* group.

Navigate to the *ansible/templates/* directory and open the *developers.j2* file in your editor. The *.j2* suffix on the file tells Ansible that it's a Jinja2 template. The contents of the file should look like this:

```
# Command alias
Cmnd_Alias       START_GREETING    = /bin/systemctl start greeting , \
                                     /bin/systemctl start greeting.service
Cmnd_Alias       STOP_GREETING     = /bin/systemctl stop greeting , \
                                     /bin/systemctl stop greeting.service
Cmnd_Alias       RESTART_GREETING  = /bin/systemctl restart greeting , \
                                     /bin/systemctl restart greeting.service

# Host Alias
Host_Alias  LOCAL_VM = {{ hostvars[inventory_hostname]['ansible_default_ipv4']
['address'] }}
# User specification
%developers LOCAL_VM = (root) NOPASSWD: START_GREETING, STOP_GREETING, \
                       RESTART_GREETING, \
                       sudoedit {{ greeting_application_file }}
```

The file begins with three `Cmnd_Alias` declarations that stop, start, and restart the Greeting web application. (In `systemd`, a service can be referred to as either `greeting` or `greeting.service`, so this handles both cases.) Next, a `Host_Alias` called `LOCAL_VM` is set to the private IP address of the VM. The built-in Ansible variable `hostvars` dynamically fetches the IP address of the VM during provision runtime. This is useful if you are provisioning many hosts at the same time. Finally, this creates a user specification for the *developers* group. (The `%` denotes it is a group and not a user.) The user specification rule states that anyone in the *developers* group, on the `LOCAL_VM`, can start, stop, restart, and edit the Greeting web application without a password, as the *root* user. Notice that issuing the `sudoedit` command is allowed only for

editing the web application. (I'll discuss `sudoedit` in more detail later.) The `{{ greeting_application_file }}` variable will be set during runtime to point to your Greeting web application file via `set_fact`.

At this point, it is safe to close all open files. Next, you'll configure the VM and test *bender*'s `sudo` privileges.

Provisioning the VM

To run all the tasks for this chapter, you need to uncomment them in the playbook like you did in previous chapters. Open the *ansible/site.yml* file in your editor and locate the task for installing the web application. It should look like this:

```
#- import_tasks: chapter4/web_application.yml
```

Remove the # symbol to uncomment it.

Next, find the task that creates the *developers sudoer* policy:

```
#- import_tasks: chapter4/sudoers.yml
```

Uncomment that line by removing the # symbol as well.

The playbook should now look like this:

```
---
- name: Provision VM
  hosts: all
  become: yes
  become_method: sudo
  remote_user: ubuntu
  tasks:
    - import_tasks: chapter2/pam_pwquality.yml
    - import_tasks: chapter2/user_and_group.yml
    - import_tasks: chapter3/authorized_keys.yml
    - import_tasks: chapter3/two_factor.yml
    - import_tasks: chapter4/web_application.yml
    - import_tasks: chapter4/sudoers.yml
  --snip--
  handlers:
    - import_tasks: handlers/restart_ssh.yml
```

The changes to the playbook for Chapter 4 are added to the changes from Chapter 3.

Now, you'll run the Ansible tasks using Vagrant. Navigate back to the *vagrant/* directory where your *Vagrant* file is located and enter the following command to provision the VM:

```
$ vagrant provision
--snip--
PLAY RECAP *********************************************************************
default       : ok=21  changed=6   unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

The values from the provision output will vary, depending on how many times you run the `provision` command, as Ansible makes sure your environment is consistent and doesn't do extra work if it's not needed. The total task count here has increased to `21`. You've also changed these six things on the VM:

- Five new tasks from Chapter 4
- One task that updates the timestamp on the empty file from Chapter 2

Once again, make sure no actions have failed before you continue.

Testing Permissions

With the VM successfully provisioned, you can now check your security policy by testing *bender*'s command access. First, you'll need to log in to the VM as *bender* again. The *sudoers* policy should allow anyone in the *developers* group (*bender*, in this case) to start, stop, restart, or edit the web application.

To log in as *bender*, grab another 2FA token. This time, locate the second 2FA token from the top in the *ansible/chapter3/google_authenticator* file; it should be `68385555`. Once you have it, enter the following command in your terminal to log in as *bender*:

```
$ ssh -i ~/.ssh/dftd -p 2222 bender@localhost
Enter passphrase for key '/Users/bradleyd/.ssh/dftd: <passphrase>
Verification code: <68385555>
--snip--
bender@dftd:~$
```

Here, you're using the SSH parameters from Chapter 3 to log in to the VM. When prompted for the 2FA token, use the second one you just grabbed. This login process should be familiar by now, but if not, revisit Chapter 3 for a refresher.

Accessing the Web Application

You'll need to make sure the web application is running and responding to requests. You'll test it with the `curl` command, which transfers data to servers (in this case, an HTTP server). The Greeting application server listens for requests on all interfaces on port 5000. So, in the terminal, enter the following command to send an `HTTP GET` request to the greeting server on port 5000:

```
bender@dftd:~$ curl http://localhost:5000
<h1 style='color:green'>Greetings!</h1>
```

The output shows the Greeting web application is responding to requests successfully on `localhost` in the VM.

Editing greeting.py to Test the sudoers Policy

Next, you'll make a small change to the Greeting application using `sudoedit` to test *bender*'s permissions. The *sudoers* policy you set earlier in this chapter allows the *developers* group members to edit the */opt/engineering/greeting.py* file with the `sudoedit` command, which lets users edit a file with any editor. It also makes a copy of the file before editing, in case things go awry. Without `sudoedit`, you might need to create multiple command aliases for each editor a user wants to use.

In a real production system, you probably would not edit a file directly on a host. Instead, you would edit the source-controlled version and allow your automation to update it with the newest version. However, I'm describing this approach to show how to test your *sudoers* policy.

While still logged in as *bender*, enter the following command to edit the *greeting.py* file:

```
bender@dftd:~$ sudoedit /opt/engineering/greeting.py
```

The command should drop you into the Nano text editor (default for Ubuntu). Once there, locate the line that looks like this inside the `hello()` function:

```
return "<h1 style='color:green'>Greetings!</h1>"
```

Change the `Greetings!` text inside the heading tag to **`Greetings and Salutations!`** so the line looks like this:

```
return "<h1 style='color:green'>Greetings and Salutations!</h1>"
```

Save the file and exit the Nano text editor.

NOTE *Feel free to use a different editor, such as Vim, if you prefer. Just be sure to set the `EDITOR` environment variable (**`export EDITOR=vim`**) before using the `sudoedit` command.*

Stopping and Starting with systemctl

For the Greeting string changes to take effect, you'll need to stop and start the web application server using `sudo` and the `systemctl` command (the latter of which is a command line application that allows you to control a service governed by `systemd`). The `Cmnd_Alias` declarations in your *sudoers* policy allow anyone in the *developers* group to run `/bin/systemctl stop greeting` or `/bin/systemctl start greeting`.

To stop the already running Greeting application using `systemctl`, enter the following command:

```
bender@dftd:-$ sudo systemctl stop greeting
```

There should be no output from the command, and you should not be prompted for a password.

Next, run curl again to be sure the web application is stopped:

```
bender@dftd:~$ curl http://localhost:5000
curl: (7) Failed to connect to localhost port 5000: Connection refused
```

Here, curl responded with a Connection refused error since the server is not running any longer.

Restart the stopped Greeting server by entering this command:

```
bender@dftd:-$ sudo systemctl start greeting
```

There won't be any output from this command if it is successful.

Run the curl command again to check whether the web application is running with the new code changes:

```
bender@dftd:~$ curl http://localhost:5000
<h1 style='color:green'>Greetings and Salutations!</h1>
```

The Greeting server provides a successful response with the new and improved greeting. If, for some reason, your Greeting application isn't responding like this, go back and retrace your steps. Start by looking for errors in the */var/log/syslog* file or the */var/log/auth.log* file on the VM.

Audit Logs

As mentioned previously, one great feature of sudo is that it leaves behind an audit trail. The events in this trail are typically used in a monitoring framework or when doing forensics during an incident response. No matter what, you should make sure the audit data is in an accessible area so you can review it.

If you followed along with the testing in this chapter, you ran the sudo command three different times. Those events were captured in the */var/log/auth.log* file, so let's explore some of the log lines from those sudo commands. I have cherry-picked a few that are pertinent to this example so you won't get bogged down in the art of log parsing. However, feel free to explore the logfile in more depth on your own.

The first line in *auth.log* you'll look at pertains to *bender*'s use of sudoedit:

```
Jul 23 23:17:43 ubuntu-focal sudo:   bender : TTY=pts/0 ; PWD=/home/bender ; USER=root ;
COMMAND=sudoedit /opt/engineering/greeting.py
```

This line provides quite a bit of information, but let's focus on the date/time, USER, and COMMAND columns. You can see that *bender* invoked sudo on July 23 at 23:17:43, using the sudoedit /opt/engineering/greeting.py command. This happened when you changed the *greeting.py* file to alter the greeting text.

This log line shows when you used *bender* to stop the Greeting server:

```
Jul 23 23:18:19 ubuntu-focal sudo:   bender : TTY=pts/0 ; PWD=/home/bender ; USER=root ;
COMMAND=/usr/bin/systemctl stop greeting
```

On July 23 at 23:18:19, *bender* used sudo to execute the /bin/systemctl stop greeting command as the *root* user.

Finally, here is the log line showing *bender* starting the Greeting application:

```
Jul 23 23:18:39 ubuntu-focal sudo:   bender : TTY=pts/0 ; PWD=/home/bender ; USER=root ;
COMMAND=/usr/bin/systemctl start greeting
```

On July 23 at 23:18:39, *bender* used sudo to execute the command /bin/systemctl start greeting as the *root* user.

So far, I have shown log entries that were successful and expected. The following line shows *bender* executing an unsuccessful command:

```
Jul 23 23:25:14 ubuntu-focal sudo:   bender : command not allowed ; TTY=pts/0 ; PWD=/home/
bender ; USER=root ; COMMAND=/usr/bin/tail /var/log/auth.log
```

On July 23 at 23:25:14, *bender* tried to run the /usr/bin/tail /var/log/auth.log command, and it was denied. These are the types of log lines you probably want to track in an alerting system, as this could be a bad actor trying to navigate a host.

NOTE *The* auth *log requires elevated permissions to read, and since your* sudoers *policy does not grant that to* bender*, you'll need to issue* sudo *as the* vagrant *user to view it.*

Summary

This chapter explored the importance of allowing users to run commands with elevated privileges. Using Ansible, the sudo command, and a *sudoers* file, you can restrict command access and log an audit trail for security. You also worked with some different Ansible modules like template, systemd, and set_fact, which allowed you to automate the installation of your web application and control its life cycle.

In the next chapter, you'll wrap up this section on provisioning and security. You'll also use some provided Ansible tasks to secure the network and implement a firewall for the VM.

5

AUTOMATING AND TESTING A HOST-BASED FIREWALL

It would be dangerous for a production server, especially one exposed to the internet, to not filter its network traffic. As software or DevOps engineers, we open up ports for services like SSH or web servers as a necessary, accepted risk. However, that does not mean we should ignore all other traffic destined for our host. To minimize risks, we need to filter all other traffic and make pragmatic decisions on what gets in and what gets out. Therefore, we use *firewalls* to monitor the incoming and outgoing packets on a network or host. Firewalls come in two varieties. A *network firewall* is usually an appliance through which all traffic flows from one network to another, while a *host-based firewall* controls the packets coming in and out of a single host.

In this chapter, you'll focus on host-based firewalls. You'll learn how to automate a host-based firewall using Ansible, some provided tasks, and a software application called Uncomplicated Firewall (UFW). This firewall will block all inbound traffic except SSH connections and the Greeting web application you installed in Chapter 4. By the end of this chapter, you'll understand how to automate a basic host-based firewall and be able to audit log events from the firewall.

Planning the Firewall Rules

Firewall rules need to be very explicit about what traffic to permit and what traffic to deny. If you accidentally block a port (or worse, leave one exposed), the outcome will be less than desirable.

You can divide the firewall traffic flow into three default parts, called *chains*. Think of a chain as a door through which a packet must pass. Each door leads to a specific place when properly routed packets arrive. Here are brief descriptions of the functions of the three default chains that you have access to in UFW:

Input chain Filters packets destined for the host

Output chain Filters packets originating from the host

Forward chain Filters packets that are being routed through the host

The firewall rules you'll create will only be for the input chain, because you're focusing on the inbound traffic to your VM. The forward and output chains are beyond the scope of this book, as you are building a simple host-based firewall. If you need to block outgoing ports and forward network traffic, visit *https://ubuntu.com/server/docs/security-firewall/* for more details.

The firewall rules you'll implement will allow incoming traffic for two known ports while rejecting all others. You'll need to open port `22` for shell access (SSH) and Ansible provisioning; plus, you'll open port `5000` for the web application. You'll also add rate limiting to port `5000`, to protect the web server and host from excessive abuse. Finally, you'll enable the firewall log so you can audit the network traffic that comes through the firewall on the VM.

Automating UFW Rules

Uncomplicated Firewall (UFW) is a software application that provides a thin wrapper around the iptables framework, which is the root of kernel-based packet filtering for Unix OSs. To be specific, iptables, Netfilter, connection tracking, and network address translation (NAT) make up the packet-filtering framework. UFW hides the complexity associated with using iptables. Along with Ansible, it makes setting up a host-based firewall simple, easy, and repeatable. Therefore, you'll use Ansible tasks to create rules with UFW.

The Ansible tasks to configure the firewall are located under the *ansible/chapter5/* directory. These rules will go into effect once you provision the VM, so let's review them before provisioning. Navigate to the *ansible/chapter5/* directory and open the task file named *firewall.yml* in your favorite editor. This file has the following five tasks in it:

1. Turn `Logging` level to low.
2. Allow SSH over port `22`.
3. Allow all access to port `5000`.
4. Rate limit excessive abuse on port `5000`.
5. Drop all other traffic.

The first task at the top of the file should look like this:

```
- name: Turn Logging level to low
  ufw:
    logging: 'low'
```

This task turns on `logging` for UFW and sets the log level to `low`. The Ansible `ufw` module creates rules and policies for the firewall on the VM. You can set the `logging` parameter to `off`, `low`, `medium`, `high`, or `full`. The `low` log level will log any blocked packets that do not match your default policy and any other firewall rules you have added. The `medium` level does everything the `low` level does, plus it logs all allowed packets that do not match the default policy and all new connections. The `high` log level does everything the `medium` does, but it also logs all packets with some rate limiting of the messages. If you have a lot of disk space and want to know everything possible about every packet on your host, set the log level to `high`. Any setting above `medium` will generate a lot of log data and could fill up disks fast on a busy host, so be careful with those log settings.

Next, let's look at the second task from the top, which opens port `22` for SSH connections. It should look like this:

```
- name: Allow SSH over port 22
  ufw:
    rule: allow
    port: '22'
    proto: tcp
```

Here, the Ansible `ufw` module creates a `rule` that allows an incoming connection from any source IP address, using the TCP transport protocol to port `22` on the VM. You can set the `rule` parameter to `deny`, `limit`, or `reject`, depending on your use case. For example, if you want to stop a connection on a specific port but don't mind sending a rejection reply to the remote host, you should choose `reject`. The rejection reply will tell the remote system that you are up and running but not accepting traffic on that port. On the other hand, if you want to drop the incoming packet on the floor without any reply to the remote host, choose a `deny` rule. This can make it

harder for someone scanning your host to know if the host is up and running. (I'll discuss the `limit` rule in detail later.)

The next task is the rule to allow remote connections on port `5000` to the Greeting web application. It should look like this:

```
- name: Allow all access to port 5000
  ufw:
    rule: allow
    port: '5000'
    proto: tcp
```

This `rule` behaves the same as the previous task, except that it permits port `5000` over TCP instead of port `22`.

The fourth task in the file limits the number of connections to port `5000` (Greeting server) over a given time frame. This is useful when you want to automatically stop someone from abusing your service, whether they are legitimate or suspicious. It should look like this:

```
- name: Rate limit excessive abuse on port 5000
  ufw:
    rule: limit
    port: '5000'
    proto: tcp
```

The default rate-limiting feature for UFW states it will deny any connection from a source if that source tries to make more than six connections in a 30-second time span. This is helpful if you host a public service like an API or web server. You could use the limit to temporarily impede users from obsessively hitting your service. Another example where this would be beneficial is to limit brute-force attempts over SSH on a *bastion host*, which is a hardened host that system administrators use to remotely access a private network. However, be careful with this default limit setting, as it may be too restrictive for a production setting. Allowing a remote system to connect more than six times in 30 seconds might be normal traffic for you. You'll test the rate-limiting rule later in this chapter.

If you want to adjust the default rate limit setting, create a new task using the `lineinfile` module (see Chapter 3) to locate and update the line in */etc/ufw/user.rules* that looks like this:

```
-A ufw-user-input -p tcp --dport 5000 -m conntrack --ctstate NEW -m recent --update --seconds
30 --hitcount 6 -j ufw-user-limit
```

Change the `hitcount` and `seconds` options to whatever makes sense for your environment.

The last task in this file drops all traffic that has not matched any other rules up to this point. Remember, Ansible executes the tasks in order. The drop rule should look like this:

```
- name: Drop all other traffic
  ufw:
    state: enabled
```

```
    policy: deny
    direction: incoming
```

Notice that there is no `rule` parameter here. This task sets the `state` of the `ufw` service to be enabled on the VM. It also sets the default `incoming` policy to `deny`, which forces you to whitelist all the services that need to be exposed. This also protects you if someone accidentally misconfigures a service and opens up a port on the host.

As mentioned previously, Ansible reads tasks from the top down, and UFW rules are read in the same order. If the `drop` rule were the first task in the file, it would set the policy to drop all traffic and then turn on the firewall. That `drop` rule would match all inbound packets and drop them, stopping the search of any other rules that possibly could match. Not only would you lose access to the VM, but you would also drop the connection made by Ansible over SSH. This means the provisioning would fail and potentially leave the machine in a bad state, so be sure to keep the order in mind when adding or removing rules.

Provisioning the VM

To run all the tasks for this chapter, you'll need to uncomment them in the playbook. This is the same process as in the previous chapters and should be familiar by now. Open the *ansible/site.yml* file in your editor and locate the task for installing the `firewall`. It should look like this:

```
#- import_tasks: chapter5/firewall.yml
```

Remove the `#` symbol to uncomment it. The playbook should now look like this:

```
---
- name: Provision VM
  hosts: all
  become: yes
  become_method: sudo
  remote_user: ubuntu
  tasks:
    - import_tasks: chapter2/pam_pwquality.yml
    - import_tasks: chapter2/user_and_group.yml
    - import_tasks: chapter3/authorized_keys.yml
    - import_tasks: chapter3/two_factor.yml
    - import_tasks: chapter4/web_application.yml
    - import_tasks: chapter4/sudoers.yml
    - import_tasks: chapter5/firewall.yml
  --snip--
  handlers:
    - import_tasks: handlers/restart_ssh.yml
```

The changes to the playbook for Chapter 5 are added on to the changes from Chapter 4.

Now, it's time to run the Ansible tasks using Vagrant. Navigate back to the *vagrant/* directory where your *Vagrantfile* is located and enter the following command to provision the VM:

```
$ vagrant provision
--snip--
PLAY RECAP *********************************************************************
default       : ok=26   changed=6    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

The total task count has increased to `26`, and `6` things on the VM have changed: the five new tasks from this chapter and one task that updates the timestamp on the empty file from Chapter 2. Once again, make sure no actions failed before you continue.

Testing the Firewall

Next, you'll want to test that your host-based firewall is enabled, permitting the two whitelisted ports, blocking all other ports, and rate-limiting the Greeting application.

First, you'll need to be able to access the VM from your local host, so grab an IP address from your VM. In the *Vagrantfile*, you told Vagrant to create another interface and let VirtualBox give it an address from a range using DHCP.

If you are no longer logged in to the VM, log in as *bender* again and grab another 2FA token, if needed. This time, grab the third 2FA token from the top of the *ansible/chapter3/google_authenticator* file, which should be `52973407`. Once you have it, enter the following command in your terminal to log in as *bender*:

```
$ ssh -i ~/.ssh/dftd -p 2222 bender@localhost
Enter passphrase for key '/Users/bradleyd/.ssh/dftd: <passphrase>
Verification code: <52973407>
--snip--
bender@dftd:~$
```

Next, use the **`ip`** command to grab the IP address from the interface you instructed Vagrant and VirtualBox to create. This command is primarily used to list and manipulate network routes and devices on a Linux host. From the VM terminal, enter the following:

```
bender@dftd:~$ ip -4 -br addr
lo               UNKNOWN        127.0.0.1/8
enp0s3           UP             10.0.2.15/24
enp0s8           UP             172.28.128.3/24
```

The output above shows that the `ip` command has completed successfully. The `-4` flag limits the output to only IPv4 addresses. The `-br` flag prints just the basic interface information, like IP address and name, and the

addr command tells ip to show the address information for the network interfaces.

The output lists three devices in tabular format. The first device, named lo, is a loopback network interface that is created on Linux hosts (commonly referred to as localhost). The loopback device is not routable (accessible) from outside the VM. The second device, enp0s3, has an IP address of 10.0.2.15. This is the default interface and the IP you get from Vagrant and VirtualBox when you first create the VM. This device is also not routable from outside the VM. The last interface, enp0s8, has an IP address of 172.28.128.3, which was dynamically assigned by this line in the *Vagrantfile*:

```
config.vm.network "private_network", type: "dhcp"
```

This IP address is how you'll access the VM from your local machine. Because these IP addresses are assigned using DHCP, yours may not match exactly. The interface name may be different as well; just use whatever IP address is listed for the interface that is not a loopback device or the device in the 10.0.2.0/24 subnet.

Keep this terminal and connection open to the VM, as you'll use it again in the next section.

Scanning Ports with Nmap

To test that the firewall is filtering traffic, you'll use the nmap (network mapper) command line tool for scanning hosts and networks. Be sure to install the appropriate Nmap version for your specific OS. Visit *https://nmap.org/book/install.html* for instructions on installing Nmap for different OSs.

Once it's installed, you'll want to do a couple of scans. The first scan, which is a fast check, tests that the firewall is enabled and allowing traffic on your two ports. The other scan is a check for the services and versions running behind those open ports.

To run the first scan, enter the following command in your terminal, using the IP address of the VM you copied earlier (if you are on a Mac or Linux host, you'll need to use sudo since Nmap requires elevated permissions):

```
$ sudo nmap -F <172.28.128.3>
Password:
Starting Nmap 7.80 ( https://nmap.org ) at 2022-08-11 10:14 MDT
Nmap scan report for 172.28.128.3
Host is up (0.00066s latency).
Not shown: 98 filtered ports
PORT     STATE SERVICE
22/tcp   open  ssh
5000/tcp open  upnp
MAC Address: 08:00:27:FB:C3:AF (Oracle VirtualBox virtual NIC)
Nmap done: 1 IP address (1 host up) scanned in 1.88 seconds
```

The `-F` flag tells `nmap` to do a fast scan, which looks for only the 100 most common ports, such as `80` (web), `22` (SSH), and `53` (DNS). As expected, the output shows `nmap` detects that ports `22` and `5000` are open. It shows the other 98 ports are *filtered*, which means `nmap` could not detect what state the ports were in because of the firewall. This tells you that the host-based firewall is enabled and filtering traffic.

The next scan you'll do is one that bad actors do on the internet every day. They scan for hosts that are connected to the internet, looking for services and versions while hoping they can match a vulnerability to it. Once they have an exploit in hand, they can use it to try to gain access to that host.

Enter the following command from your local host's terminal to detect your service versions:

```
$ sudo nmap -sV <172.28.128.3>
Starting Nmap 7.80 ( https://nmap.org ) at 2022-08-11 21:06 MDT
Nmap scan report for 172.28.128.3
Host is up (0.00029s latency).
Not shown: 998 filtered ports
PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.1 (Ubuntu Linux; protocol 2.0)
5000/tcp open  http    Gunicorn 20.0.4
MAC Address: 08:00:27:F7:33:1F (Oracle VirtualBox virtual NIC)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/.
Nmap done: 1 IP address (1 host up) scanned in 13.13 seconds
```

The `-sV` flag tells `nmap` to attempt to extract service and version information from running services. Once again, `nmap` finds the two open ports, `22` and `5000`. Also, a service name and version are listed next to each port. For port `22`, the service name is `OpenSSH`, and the version is `8.2p1` for `Ubuntu Linux`. For port `5000`, the service name is `Gunicorn`, and the version is `20.0.4`. If you were a bad actor armed with this information, you could search the many vulnerability databases, looking for exploits for these services and versions.

NOTE *Nmap is something all software and DevOps engineers should have in their tool belts. Learn more about Nmap from the man page or* https://nmap.org/.

Next, you'll want to check the logs for evidence that the firewall blocked connection attempts on non-whitelisted ports.

Firewall Logging

All events that the firewall processes can be logged. You enabled logging and set the level to `low` for UFW in the Ansible task earlier in this chapter. The log for those events is located in the */var/log/ufw.log* file. This logfile requires *root* permissions to read it, so you'll need a user with elevated permissions.

As an example, I have pulled out a log entry to demonstrate a block event from the *ufw.log* file. Here is what UFW logged when Nmap tried to scan port 80:

```
Aug 11 16:56:17 ubuntu-focal kernel: [51534.320364] ❶[UFW BLOCK] ❷IN=enp0s8
OUT= MAC=08:00:27:fb:c3:af:0a:00:27:00:00:00:08:00 ❸SRC=172.28.128.1
❹DST=172.28.128.3 LEN=44 TOS=0x00 PREC=0x00 TTL=48 ID=7129 PROTO=TCP
SPT=33405 ❺DPT=80 WINDOW=1024 RES=0x00 SYN URGP=0
```

This log line contains a lot of information, but you'll focus on only a few components here. The event type name ❶ is a block type, so it's named `[UFW BLOCK]`. The `IN` key-value pair ❷ shows the network interface for which this packet was destined. In this case, it's the VM interface from the earlier section. The source IP address (`SRC`) ❸ is where the packet originated. In this example, it's the source IP address from the local host where you ran the `nmap` command. This IP address was created from VirtualBox when you added the other interface in Vagrant. The destination IP address, `DST` ❹, is the IP address for which the packet was destined. It should be the IP address of the second non-loopback interface on the VM. The destination port, `DPT` ❺, is the port where the packet was being sent. In this log line, it's port `80`. Since you don't have a rule permitting any traffic on port `80`, it was blocked. This means your firewall is blocking unwanted connection attempts. Remember, Nmap's fast scan will try 100 different ports, so there will be multiple log lines that look like this one. However, they will have different destination ports (`DPT`).

Rate Limiting

To test that the firewall will rate-limit excessive connection attempts (six in 30 seconds) to your Greeting web server, you'll leverage the `curl` command again. From your local host, enter the following to access the Greeting web server six times:

```
$ for i in `seq 1 6` ; do curl -w "\n" http://172.28.128.3:5000 ; done
<h1 style='color:green'>Greetings!</h1>
<h1 style='color:green'>Greetings!</h1>
<h1 style='color:green'>Greetings!</h1>
<h1 style='color:green'>Greetings!</h1>
<h1 style='color:green'>Greetings!</h1>

curl: (7) Failed to connect to 172.28.128.22 port 5000: Connection refused
```

Here, a simple `for` loop in Bash iterates and executes the `curl` command six times in succession. The `curl` command uses the `-w "\n"` flag to write out a new line after each loop, which makes the web server's response output more readable. As you can see, the last line shows a `Connection refused` notification after the fifth successful connection to the Greeting web server. This is because the rate limit on the firewall for port `5000` was triggered by being hit six times in less than 30 seconds.

Let's explore the log line for this event. (Once again, I've grabbed the relevant log line for you.)

```
Aug 11 17:38:48 ubuntu-focal kernel: [54085.391114] ❶ [UFW LIMIT BLOCK]
IN=enp0s8 OUT= MAC=08:00:27:fb:c3:af:0a:00:27:00:00:00:08:00
❷SRC=172.28.128.1 ❸DST=172.28.128.3 LEN=64 TOS=0x00 PREC=0x00 TTL=64 ID=0 DF
PROTO=TCP SPT=58634 ❹DPT=5000 WINDOW=65535 RES=0x00 CWR ECE SYN URGP=0
```

The UFW event type is named `[UFW LIMIT BLOCK]` ❶. This packet is coming (`SRC`) from the local host IP address ❷ where you ran the `curl` command. The destination (`DST`) ❸ IP address is the one for the VM. The destination port (`DPT`) ❹ is `5000`, which is the Greeting web server. This temporary limit will block your local host IP address (`172.28.128.1`)❷ from accessing port `5000` for about 30 seconds after the limit is reached. After that, you should be able to access it again.

Summary

In this chapter, you've learned how to implement a simple but effective host-based firewall for the VM. You can easily apply this firewall to any host you have, whether it is local or from a cloud provider. Creating firewall rules with Ansible that permit specific traffic to a VM while blocking other traffic is a typical setup a DevOps or software engineer would use. You also learned how to limit the number of connections a host can make in a given time frame. All of these techniques provide a smaller attack surface to help deter network attacks. You can do a lot more to enhance your host-based firewall, and I encourage you to explore the possibilities on your own by visiting *https://help.ubuntu.com/community/UFW/*.

This brings Part I to an end. You now should have a good understanding of how to provision your infrastructure and apply some basic security foundations to your environment. In Part II, we'll move on to containers, container orchestration, and deploying modern application stacks. We'll start with installing and understanding Docker.

PART II

CONTAINERIZATION AND DEPLOYING MODERN APPLICATIONS

6

CONTAINERIZING AN APPLICATION WITH DOCKER

A *container* is the running instance of an application based off a container image. Using containers provides you with a predictable and isolated way to create and run code. It allows you to package an application and its dependencies into a portable artifact you can easily distribute and run. Microservice architectures and continuous integration/continuous development pipelines heavily use containers, and if you're a software or DevOps engineer, using containers has most likely changed the way you deliver and write software.

In this chapter, you'll learn how to install the Docker engine and the `docker` client command line tool. You'll also get a crash course in Dockerfiles, container images, and containers. You'll combine this knowledge, along with some basic Docker commands, to containerize a sample

application called *telnet-server* that I've provided in the repository for this book (*https://github.com/bradleyd/devops_for_the_desperate/*). By the end of this chapter, you'll have a solid understanding of how to use Docker to containerize any application, as well as the benefits of doing so.

Docker from 30,000 Feet

The word *Docker* has become synonymous with the container movement. This is due to Docker's ease of use, the rise of microservice architectures, and the need to solve the "works on my machine" paradox. The idea of containers has been around for quite some time, however, and numerous container frameworks exist. But since Docker released its first open-source version in March 2013, the industry has adopted the Docker framework as the de facto standard. The first stable version of Docker (1.0) was released in 2014, and since then, new versions have included many improvements.

The Docker framework consists of a Docker daemon (server), a `docker` command line client, and other tools that are beyond the scope of this book. Docker uses Linux kernel features to build and run containers. These pieces fit together to allow Docker to do its magic: *OS-level virtualization*, which partitions the operating system into what looks like separate isolated servers, as shown in Figure 6-1. Because of this, containers are effective when you need to run a lot of applications on limited hardware.

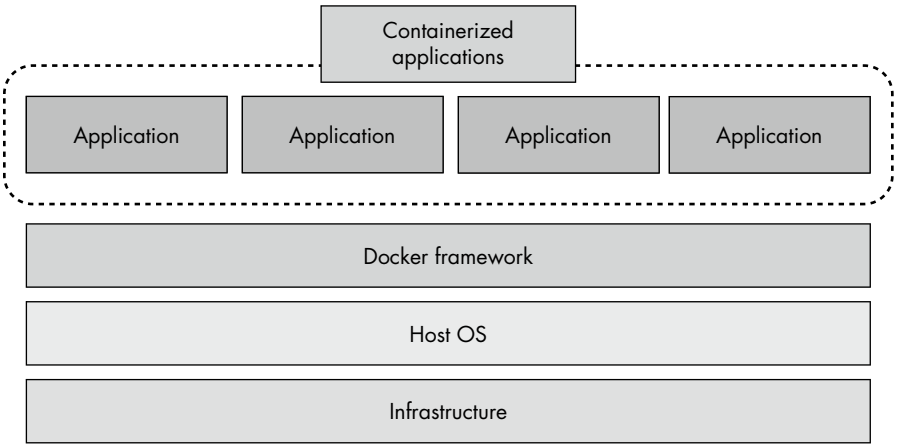


Figure 6-1: OS-level virtualization

Getting Started with Docker

First, you'll create a *Dockerfile* that describes how to build the *container image* from your application. A container image is made of different layers that house your application, dependencies, and anything else the application needs so it can run. Container images can be distributed and served from a service called a *registry*. Docker hosts the most popular registry at

https://hub.docker.com/. There, you'll find just about any image you might need, such as Ubuntu or the PostgreSQL database. With a simple `docker pull <image-name>` command, you can download and use an image in a matter of seconds. A container is the running instance of an application based off the container image. Figure 6-2 shows how all of Docker's pieces fit together. In this chapter, you'll mostly be working with the `docker` client.

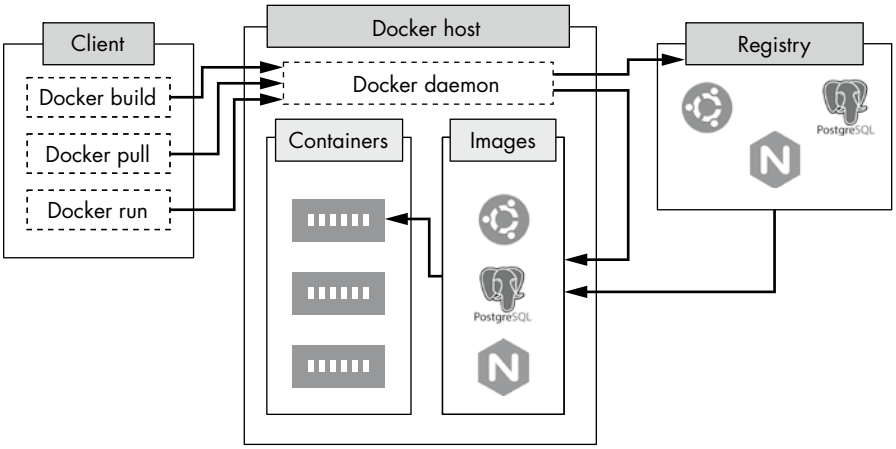


Figure 6-2: Docker framework

Dockerfile Instructions

The Dockerfile contains the instructions that teach the Docker server how to turn an application into a container image. Each instruction represents a specific job and creates a new layer inside the container image. The following list includes the most common instructions:

`FROM` Specifies the parent or base image from which to build the new image (must be the first command in the file)

`COPY` Adds files from your current directory (where the Dockerfile resides) to a destination in the image filesystem

`RUN` Executes a command inside the image

`ADD` Copies new files or directories from either a source or a URL to a destination in the image filesystem

`ENTRYPOINT` Makes your container run like an executable (which you can think of as any Linux command line application that takes arguments on your host)

`CMD` Provides a default command or default parameters for the container (can be used in conjunction with `ENTRYPOINT`)

See the Dockerfile reference at *https://docs.docker.com/engine/reference/builder/* for instructions and configuration details.

Container Images and Layers

The Dockerfile you build creates a container image. This image is made of different layers that house your application, dependencies, and anything else the application needs so it can run. These layers are like snapshots in time of your application's state, so keeping your Dockerfiles in version control along with your source code makes it easier to build new container images every time your application code changes.

The layers fit together like LEGO bricks. Each layer, or intermediate image, is created each time an instruction in the Dockerfile is executed. For example, every time you use the `RUN` instruction, a new intermediate layer is created with the results of that instruction. Each layer (image) is assigned a unique hash, and all layers are cached by default. This means you can share layers with other images, so if a given layer hasn't changed, you don't need to build it again from scratch. Also, caching is your best friend, as it cuts down the time and space needed to build images.

Docker can stack these layers on top of each other because it uses the *union filesystem (UFS)*, which allows multiple filesystems to come together and create what looks like a single filesystem. The topmost layer is the *container layer*, which is added when you run the container image. It's the only layer that can be written to. All the subsequent layers are read only, by design. If you make any file or system changes to the container layer and then remove the running container, those changes will be gone. The underlying read-only images are kept intact. This is why containers are so popular with software engineers: the image is an immutable artifact that can be run on any Docker host and behave in the same way.

Containers

The Docker container is a running instance of a container image. In computer programming terms, you can think of the container image as a *class* and the container as an *instance* of that class. When the container starts, the container layer is created. This writeable layer is where all the changes (like writing, deleting, and modifying existing files) will take place.

Namespaces and Cgroups

The container is also roped off from the rest of the Linux host by some boundaries and limited views called *namespaces* and *cgroups*. These are kernel features that limit what a container can see and use on a host. They also make OS-level virtualization a reality. Namespaces restrict global system resources for a container. Without namespaces, a container could have free run of the system. Imagine if a container could see a process in another container. That mischievous container could kill a process, delete a user, or unmount a directory in another container. Try tracking that down when you're on call at 2 AM!

Common kernel namespaces include the following:

Process ID (`PID`) Isolates the process IDs

Network (`net`) Isolates the network interface stack

UTS Isolates the hostname and domain name

Mount (`mnt`) Isolates the mount points

IPC Isolates the SysV-style interprocess communication

User Isolates the user and group IDs

Using these namespaces is not enough, however. You also need to control how much memory, CPU, and other physical resources a container uses. That's where cgroups come in. Cgroups manage and measure the resources a container can use. They allow you to set resource limitations and prioritization for processes. The most common resources Docker sets with cgroups are memory, CPU, disk I/O, and network. Cgroups make it possible to stop a container from using up all the resources on a host.

The main point to remember is that namespaces limit what you can see, while cgroups limit what you can use. Without these features, containers would not be secure or useful.

Installing and Testing Docker

To containerize a sample application, you'll start by installing Docker with the aid of *minikube*, an app that contains the Docker engine and also provides a Kubernetes cluster (which you'll use in the next chapter). Next, you'll install the `docker` client so that you'll be able to communicate with the Docker server. Then, you'll configure your environment so that it can find the new Docker server. Finally, you'll test client connectivity.

Installing the Docker Engine with Minikube

To install minikube, follow the instructions for your operating system at *https://minikube.sigs.k8s.io/*. If you're not on a Linux host, minikube requires a virtual machine manager to install Docker. Use VirtualBox for that.

By default, minikube makes a best guess about memory allocation for the VM it will create. It also sets the number of CPUs to two and the disk space to 20GB. For the purposes of this book, the defaults should be fine.

> **OVERRIDING MINIKUBE'S DEFAULTS**
>
> Pass the `--cpus=<` *number*`>`, `--memory='<` *number*`>'`, and `--disk-size='<` *number*`>'` arguments to the `minikube start` command to change the defaults. Be sure to include the appropriate unit. For example, you could enter `minikube start --cpus=4 --memory='10g' --disk-size='40g'` to give minikube more resources.

To start minikube using the resource defaults and VirtualBox as the VM manager, enter the following in a terminal:

```
$ minikube start --driver=virtualbox
 --snip--
 Done! kubectl is now configured to use "minikube"
```

The `Done!` message shows that minikube started successfully. If minikube fails to start, you should investigate any error messages listed in the output.

Installing the Docker Client and Setting Up Docker Environment Variables

To install the `docker` client, follow the instructions at *https://docs.docker.com/engine/install/binaries/* for your operating system. Make sure you only download and install the client binary. You'll use minikube to set some local environment variables in your shell, including the Docker host IP and the path to the Docker host TLS certificates, which are needed to connect. The Bash `eval` command will source the environment variables in your shell.

In a terminal, enter the following to set your Docker environment variables:

```
$ eval $(minikube -p minikube docker-env)
```

This command should return zero output if it's successful. The Docker host environment variables should be exported in your current terminal session.

When you close this terminal window, the environment variables will be lost, and you'll need to run the command each time you want to interact with the Docker server. To avoid this inconvenience, add the command to the bottom of your shell configuration file such as *~/.bashrc* or *~/.zshrc* so it's executed each time you open a terminal window or tab. Then you won't see the `Is the docker daemon running?` error.

Testing the Docker Client Connectivity

You should test whether the `docker` client can talk to the Docker server running inside the minikube VM. In the same terminal where you set the environment variables, enter the following to check the Docker version:

```
$ docker version
```

The output should show your client and server versions if the connection is successful.

Containerizing a Sample Application

I created a sample application named *telnet-server* that you can use to build a container with Docker. It's a simple telnet server that mimics the bulletin board systems (BBSs) people used in the 1980s. The app is written in the Go programming language for OS portability and a small footprint.

You'll use an Alpine Linux container image that contains Go and all the needed dependencies.

To containerize an application, you'll need the source code or binary you want to run in the container plus the Dockerfile to build the container image. The sample application source code and Dockerfile are in the companion repository for this book at *https://github.com/bradleyd/devops_for_the_desperate/* in the *telnet-server/* folder.

Dissecting the Example telnet-server Dockerfile

The example Dockerfile is a *multistage build* with two separate stages: *build* and *final*. Multistage builds allow you to manage complex builds in one Dockerfile, and they provide a good pattern for keeping container images small and secure. In the build stage, the Dockerfile instruction compiles the sample application with all its dependencies. In the final stage, the Dockerfile instruction copies the build artifact (in this case, the compiled sample application) from the build stage. The final container image is much smaller because it doesn't contain all the dependencies or source code for the sample application from the build stage. Visit *https://docs.docker.com/develop/develop-images/multistage-build/* for more information on multistage builds.

Navigate to the *telnet-server/* directory and open the Dockerfile, which should look like this:

```
# Build stage
FROM golang:alpine AS build-env
ADD . /
RUN cd / && go build -o telnet-server

# Final stage
FROM alpine:latest AS final
WORKDIR /app
ENV TELNET_PORT 2323
ENV METRIC_PORT 9000
COPY –from=build-env /telnet-server /app/
ENTRYPOINT ["./telnet-server"]
```

The file starts the build stage with a `FROM` instruction to pull in the `golang:alpine` parent image. This is an Alpine Linux image from the Docker Hub registry that's prebuilt for developing in the Go programming language. This image stage is named `build-env`, using the `AS` keyword. This name reference is used again later, in the final stage.

The `ADD` instruction copies all the Go source code in the current local *telnet-server/* directory to the image's filesystem at the root (/) destination.

The next `RUN` instruction executes the shell command that navigates to the root directory in the image filesystem, and it uses the `go build` command to build the Go binary named telnet-server.

The final stage begins with a `FROM` instruction that again pulls in an Alpine Linux image (`alpine:latest`) for the final stage's parent image. This time, though, the Alpine Linux image is the minimal image in which the application will run. It doesn't contain any dependencies.

The `WORKDIR` instruction sets the working directory for the application, which is */app* in this example. Any `CMD`, `RUN`, `COPY`, or `ENTRYPOINT` instruction after that declaration will be executed in the context of that working directory.

The two `ENV` instructions set environment variables in the container image that the application can use: they set the telnet server to port `2323` and the metric server port to `9000`. (More on those ports later.)

The `COPY` instruction copies the telnet-server Golang binary from the `build-env` stage and places it in the working */app/* directory in the final-stage Alpine image.

The final `ENTRYPOINT` instruction invokes the telnet-server binary when the container starts to execute the sample application. You'll use `ENTRYPOINT` instead of `CMD` because the application will require additional flags passed to it during a container test in a later chapter. If you need to override the default command in your container, swap `ENTRYPOINT` with the `CMD` instruction instead. See the Dockerfile reference at *https://docs.docker.com/engine/reference/builder/* to learn more about `CMD` versus `ENTRYPOINT`.

NOTE *Notice that the on-disk sizes for the `golang:alpine` and `alpine:latest` images are very different. The Go base image comes in at around 315MB, and the `alpine:latest` image is 5.59MB. Multistage builds are effective at keeping down container size, which means faster downloads, quicker startup times, and more disk space. When it comes to containers, size matters.*

Building the Container Image

Next, you'll build the container image for the sample telnet-server application, using the Dockerfile you just reviewed. Navigate to the *telnet-server/* directory and enter the following to pass Docker the image name and Dockerfile location:

```
$ docker build -t dftd/telnet-server:v1 .
```

The `-t` flag sets the name and (optionally) a tag for the image, and the dot (`.`) argument sets the Dockerfile's current location. The *dftd/telnet-server:v1* URI has three parts: the registry hostname (*dftd*), the image name, and the tag. The registry is local to minikube rather than online, so you can use anything for the base. (If it were a remote registry, you'd use something like *registry.example.com*.) The image name sandwiched between the forward slash (`/`) and the colon (`:`) is set to the name of the example application, *telnet-server*. The *v1* image tag comes after the colon.

Tags allow you to identify each build of an image and indicate what changes are inside. Using Git commit hashes as tags is a common practice, as each hash is unique and can mark the image's source code version. If you omit the tag, Docker uses the latest word as the default tag.

After running the command, you should see output like this:

```
Sending build context to Docker daemon     13MB
Step 1/9 : FROM golang:alpine AS build-env
 ---> 6f9d081b1170
```

```
Step 2/9 : ADD . /
 ---> 3146d8206747
Step 3/9 : RUN cd / && go build -o telnet-server
 ---> Running in 3e05a0704b36
go: downloading github.com/prometheus/client_golang v1.6.0
go: downloading github.com/prometheus/common v0.9.1
go: downloading github.com/prometheus/client_model v0.2.0
go: downloading github.com/beorn7/perks v1.0.1
go: downloading github.com/cespare/xxhash/v2 v2.1.1
go: downloading github.com/golang/protobuf v1.4.0
go: downloading github.com/prometheus/procfs v0.0.11
go: downloading github.com/matttproud/golang_protobuf_extensions
  v1.0.1 1 # Build stage
go: downloading google.golang.org/protobuf v1.21.0
go: downloading golang.org/x/sys v0.0.0-20200420163511-1957bb5e6d1f
Removing intermediate container 3e05a0704b36
 ---> 96631440ea5d
Step 4/9 : FROM alpine:latest AS final
 ---> c059bfaa849c
Step 5/9 : WORKDIR /app
 ---> Running in ddc5b73b1712
Removing intermediate container ddc5b73b1712
 ---> 022bcbba3b94
Step 6/9 : ENV TELNET_PORT 2323
 ---> Running in 21bd3d15f50c
Removing intermediate container 21bd3d15f50c
 ---> 30d0284cade4
Step 7/9 : ENV METRIC_PORT 9000
 ---> Running in 8f1fc01b04d5
Removing intermediate container 8f1fc01b04d5
 ---> adfd026e1c27
Step 8/9 : COPY --from=build-env /telnet-server /app/
 ---> fd933cd32a94
Step 9/9 : ENTRYPOINT ["./telnet-server"]
 ---> Running in 5d8542e950dc
Removing intermediate container 5d8542e950dc
 ---> f796da88ab94
Successfully built f796da88ab94
Successfully tagged dftd/telnet-server:v1
```

Each instruction is logged, allowing you to follow along with the build process in a linear fashion. At the end of the build, the image ID (`f796da88ab94`) should be listed, followed by a note that the image is tagged successfully as *dftf/telnet-server:v1*. The image ID you see will be different.

If your `docker build` wasn't successful, you'll want to resolve any errors in the output because you'll build upon this image going forward. Common errors are typos in the `RUN` execution and missing files when using the `COPY` instruction.

Verifying the Docker Image

Next, verify that the Docker registry inside minikube is storing the telnet-server image. (As mentioned previously, a registry is a server that stores and serves container images.)

In a terminal, enter the following to list the Docker telnet-server image:

```
$ docker image ls dftd/telnet-server:v1
REPOSITORY                  TAG         IMAGE ID         CREATED          SIZE
dftf/telnet-server          v1          f796da88ab94     1 minute ago     16.8MB
```

Notice that the final image for the telnet-server is only 16.8MB. The Alpine Linux base image in the final stage was roughly 5MB before adding the telnet-server application.

Running the Container

The next step is to create and run the telnet-server container from the image you just built. Do this by entering the following:

```
$ docker run -p 2323:2323 -d --name telnet-server dftd/telnet-server:v1
9b4b719216a1664feb096ba5a67c54907268db781a28d08596e44d388c9e9632
```

The `-p` (port) flag exposes port `2323` outside the container. (The telnet-server application needs to have port `2323` open.) The left side of the colon (`:`) is the host port, and the right side is the container port. This is useful if you have another application listening on the same port and need to change it for the host while keeping the container port the same. The `-d` (detach) flag launches the container in the background. If you don't supply the `-d` flag, the container will run in the foreground of the terminal from which it launched. The `--name` flag sets the container name to `telnet-server`. Docker, by default, assigns randomly generated names for containers if you don't set them. The last argument is the image name, complete with path and tag, from the build step.

The container is now running in the background and ready to accept traffic. This `docker run` command was successful because it returned the *container ID* (the long string of numbers and letters, which will be different for you) and no errors.

NOTE *The volume flag, `-v`, can mount a local directory or local file inside the running container. This is a great way to share data between host and container.*

Enter the following to verify that the container is actually running:

```
$ docker container ls -f name=telnet-server
```

The optional filter flag (`-f`) narrows the output to the containers you specify. If you omit the filter flag, running the command should list every container running on the host.

If the container is running, the output should look like this:

```
CONTAINER ID  IMAGE         COMMAND       ...   PORTS                   NAMES
9b4b719216a1  dftd/...      "./telnet-.." ...   0.0.0.0:2323->2323/tcp   telnet-server
```

The CONTAINER ID column matches the first 12 digits of the ID received from the docker run command issued previously. The IMAGE column contains the image ID given when you built the container image. The PORTS column shows that port 2323 is exposed on every interface (0.0.0.0) and is mapping that traffic to port 2323 inside the container. The directional arrow (->) denotes the traffic flow direction. Finally, the NAMES column shows the telnet-server name set earlier from the run command.

Now, enter the following in your terminal to stop the container:

```
$ docker container stop telnet-server
telnet-server
```

The container name should be returned, letting you know the Docker daemon thinks the container is stopped. To start the container again, swap the word stop with start, and you should see the container name returned again.

Docker won't check to see whether your application stays running after you start it. As long as the container can start and not error out immediately, entering docker start or docker run will return the container name as if nothing were wrong. This can be misleading. You'll want to perform health checks and monitor the application to verify that it's actually running. (We'll explore those topics in future chapters.)

Other Docker Client Commands

Let's look at a few more common Docker commands you'll need to use when working with containers.

exec

The exec command allows you to run a command inside a container or interact with a container, as if you were logged in to a terminal session. For example, if you are troubleshooting an application in a container and want to verify that the correct environment variables are being set, you could run the following command in a terminal to output all the environment variables:

```
$ docker exec telnet-server env
TELNET_PORT=2323
HOSTNAME=c8f66b93424a
SHLVL=1
HOME=/root
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/app
METRIC_PORT=9000
```

The env command executes inside the container, using the OS's default shell. When it finishes, the output is sent back to the terminal.

The `exec` command also allows you to access a running container to troubleshoot it or run a command. You'll need to pass the interactive flag (`-i`) and the pseudo-TTY flag (`-t`), along with the shell command (`/bin/sh`), to do this. The interactive flag keeps STDIN open so you can type commands inside the container layer. The pseudo-TTY flag simulates a terminal, and when combined with the interactive flag, it mimics being in a live terminal session inside the container. Operating systems other than Linux will use different shells: most commonly, */bin/sh* and */bin/bash*. Alpine Linux uses the */bin/sh* shell as its default.

Enter the following in a terminal to get a shell inside the container:

```
$ docker exec -it telnet-server /bin/sh
/app # ls
telnet-server
/app #
```

The `ls` command is issued to show you're inside the container you built. (You earlier set the working directory to *app/* and put the telnet-server binary in there.) Input the **`exit`** command and press ENTER to leave the container and return to the local terminal.

rm

The `rm` command removes a stopped container. For example, to remove the telnet-server container once it is stopped, enter the following in a terminal:

```
$ docker container rm telnet-server
telnet-server
```

The removed container's name should be returned. You can use the `-f` (force) flag to remove a running container, but it's best to stop it first.

inspect

The `inspect` `docker` command returns low-level information about some Docker objects. The output is in JSON format by default. Depending on the Docker object, the results can be verbose.

To inspect the telnet-server container, enter the following in a terminal:

```
$ docker inspect telnet-server
[
    {
        "Id": "c8f66b93424a3dac33415941e357ae9eb30567a3d64d4b5e87776701ad8274c5",
        "Created": "2022-02-16T03:35:44.777190911Z",
        "Path": "./telnet-server",
        "Args": [],
        "State": { ❶
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
```

```
            "Dead": false,
            "Pid": 19794,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2022-02-16T03:35:45.230788473Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
--snip--
        "NetworkSettings": { ❷
            "Bridge": "",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {
                "2323/tcp": [
                    {
                        "HostIp": "0.0.0.0",
                        "HostPort": "2323"
                    }
                ]
            },
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.5",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
         "MacAddress": "02:42:ac:11:00:05",
--snip--
```

The `State` section ❶ contains data about the running container, like `Status` and `StartedAt` date. The `NetworkSettings` section ❷ provides information like `Ports` and `IPAddress`, which are helpful when troubleshooting problematic containers.

history

The `history` command displays a container image's history, which is useful for viewing the number and sizes of an image's layers.

To see the telnet-server image's layers, enter the following in a terminal:

```
$ docker history dftd/telnet-server:v1
IMAGE          CREATED       CREATED BY                                      SIZE    COMMENT
cb5a2baff085   20 hours ago  /bin/sh -c #(nop) ENTRYPOINT ["./telnet-ser...  0B
a826cfe49c09   20 hours ago  /bin/sh -c #(nop) COPY file:47e9acb5fa56759e... 13MB
a9a45301f95b   5 days ago    /bin/sh -c #(nop) ENV METRIC_PORT=9000          0B
001a12a073c2   5 days ago    /bin/sh -c #(nop) ENV TELNET_PORT=2323          0B
379892a150e3   6 days ago    /bin/sh -c #(nop) WORKDIR /app                  0B
f70734b6a266   3 weeks ago   /bin/sh -c #(nop) CMD ["/bin/sh"]               0B
<missing>      3 weeks ago   /bin/sh -c #(nop) ADD file:b91adb67b670d3a6f... 5.61MB
```

The output (edited) shows the instructions that start each layer, like `COPY` and `ADD`. It also shows the layers' ages and sizes.

stats

The `stats` command displays a real-time update on the resources a container is using. It gathers this information from the cgroups and behaves similarly to the Linux `top` command. If you have a host that manages multiple containers and want to see which one is the resource hog, use the `stats` command. Once you run the `stats` command, it drops you into a page that updates every few seconds. As that's impossible to show in a book, we'll pass the `--no-stream` flag to take a snapshot of the resources and exit immediately.

Enter the following to show the telnet-server container's resource usage:

```
$ docker stats --no-stream telnet-server
CONTAINER ID  NAME           CPU %  MEM USAGE / LIMIT  MEM %   NET I/O  BLOCK I/O   PIDS
c8f66b93424a  telnet-server  0.00%  2.145MiB/5.678GiB  0.04%   0B / 0B  0B / 0B      7
```

This telnet-server container is using virtually no CPU, no disk or network I/O, and only 2MiB of memory. You could easily run hundreds of these in a cloud environment on a single server.

Visit *https://docs.docker.com/engine/reference/commandline/cli/* to explore all of the `docker` command line client's commands and flags.

Testing the Container

To find out whether the sample application you've containerized actually works, you'll connect to the telnet-server on port `2323` and run some basic commands. Then you'll view the container logs to verify that the application is working correctly.

Before performing either of these steps, however, you'll need to install a `telnet` client for your OS to communicate with the telnet-server. If you're using macOS, simply enter **`brew install telnet`** in your terminal. If you're using Ubuntu, enter **`apt install telnet`** in a terminal as a privileged user.

Connecting to the Telnet-Server

To connect to the server, pass telnet the hostname or IP address of the server plus the port to which you want to connect. Since the Docker server is running inside a VM (minikube), you'll need the IP address minikube exposes to your local host.

Enter the following in a terminal to get the IP address:

```
$ minikube ip
192.168.99.103
```

My minikube IP address is 192.168.99.103; yours may be different.

To connect to the telnet-server running inside the container, pass the IP address (192.168.99.103) and port (`2323`) to the `telnet` command:

```
$ telnet 192.168.99.103 2323
Trying 192.168.99.103...
Connected to 192.168.99.103.
Escape character is '^]'.


____________ __________
|  _  \  ___|_   _|  _  \
| | | | |_    | | | | | |
| | | |  _|   | | | | | |
| |/ /| |     | | | |/ /
|___/ \_|     \_/ |___/

>
```

Success! The ASCII text banner of DFTD should greet you in all its glory. You're now connected to the telnet-server application. The prompt (>) is where you can enter commands. To start, you can enter only the `date`, `help`, `yell`, and `quit` commands. You can use the first character of any of those commands as a shortcut, and any commands you enter will be logged.

While still connected to the telnet-server, enter the following to print the current date and time:

```
>d
Tue May 10 22:55:13 +0000 UTC 2022.
```

Great! The current date and time should be displayed. Depending on your age, this could bring back memories of baud rates and high-pitched squeals.

Enter the following to quit the telnet-server session:

```
>q
Good Bye!
Connection closed by foreign host.
```

You should see that the telnet-server session is nice enough to say goodbye. Take that, modern-day internet!

You can add new commands to the server or change the responses in the *telnet-server/telnet/server.go* file. If you do, don't forget to build, stop, and replace the image and container using the commands you learned earlier in this chapter.

Getting Logs from the Container

Docker provides a simple way to retrieve logs from a running container. This is important for troubleshooting and forensics purposes.

To see all the logs for the telnet-server, which is logging to STDOUT, enter the following in your terminal:

```
 $ docker logs telnet-server
telnet-server: 2022/01/04 19:38:22 telnet-server listening on [::]:2323
telnet-server: 2022/01/04 19:38:22 Metrics endpoint listening on :9000
telnet-server: 2022/01/04 19:38:32 [IP=192.168.99.1] New session
telnet-server: 2022/01/04 19:38:43 [IP=192.168.99.1] Requested command: d
telnet-server: 2022/01/04 19:38:44 [IP=192.168.99.1] User quit session
```

The first two lines of output are startup messages showing that the server is running and listening on specific ports. (We'll explore the metrics server when we look at monitoring applications in Chapter 9.) The fourth log line is from when you entered the `d` command into the telnet session to print the current date and time. The fifth log line shows when you entered `q` to exit the test telnet session.

NOTE *The `logs` command can also mimic the Linux `tail` command. Use the `-f` flag to follow the log stream or the `--tail` flag to limit the number of lines shown.*

Summary

If you're a software or DevOps engineer, you need a solid understanding of containers in today's infrastructure. In this chapter, you explored how Docker makes containers possible with OS-level virtualization. You examined how a Dockerfile works to create the layers of a container image, and you applied that knowledge to build a sample container image using a multi-stage build. Finally, you started a container from the provided telnet-server image, tested that it was working correctly, and checked its logs. In the next chapter, you'll take the telnet-server image you built here and run it inside a Kubernetes cluster.

7

ORCHESTRATING WITH KUBERNETES

A container makes applications portable and consistent, but it's only one piece of a modern application stack. Imagine needing to manage thousands of containers on different hosts, network ports, and shared volumes. What if one container stopped? How could you scale for load? How could you force containers to run on different hosts for availability? Container *orchestration* solves all these issues and more. *Kubernetes*, or *K8s*, is the open-source orchestration system many companies use to manage their containers. Kubernetes comes preloaded with some useful patterns (such as networking, role-based access control, and versioned APIs), but it's meant to be the foundational framework on which to build your unique infrastructure and tools. Kubernetes is the standard in container orchestration. You can think of it as a low-level piece of your infrastructure, just like Linux.

In this chapter, you'll learn some basic Kubernetes resources and concepts concerning container orchestration. To put orchestration into practice, you'll deploy the telnet-server container image from Chapter 6 inside your Kubernetes cluster using the `kubectl` command line client.

Kubernetes from 30,000 Feet

Kubernetes (which means *helmsman* in Greek) evolved from its predecessors, Borg and Omega, at Google. It was open-sourced in 2014 and has received great community support and many enhancements since then.

A Kubernetes cluster consists of one or more control plane nodes and one or more worker nodes. A *node* can be anything from a cloud VM to a bare-metal racked server to a Raspberry Pi. The *control plane nodes* handle things like the Kubernetes API calls, the cluster state, and the scheduling of containers. The core services (such as the API, etcd, and the scheduler) run on the control plane. The *worker nodes* run the containers and resources that are scheduled by the control plane. See Figure 7-1 for more details.

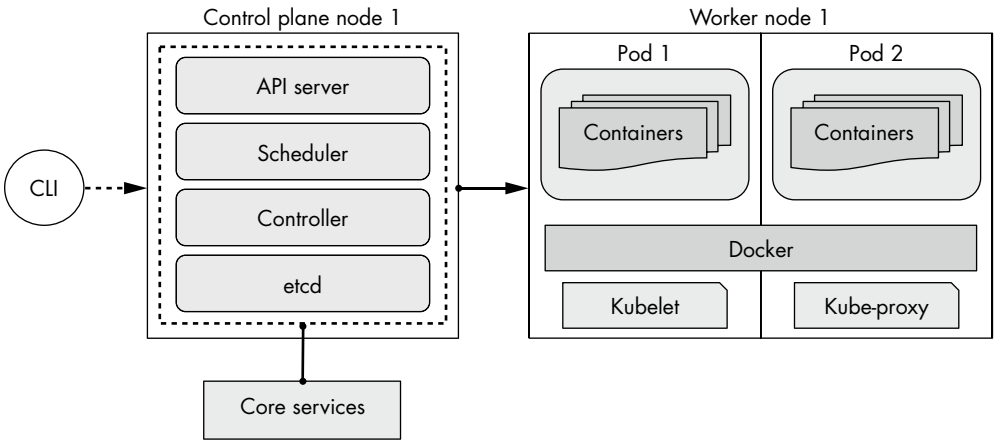


Figure 7-1: The basic building blocks of a Kubernetes cluster

Networking and scheduling are the most complex issues you'll encounter when orchestrating containers. When networking containers, you must consider all the ports and access they need. Containers can communicate with each other, both inside and outside the cluster. This happens with microservices internal communication or when running a public-facing web server. When scheduling containers, you must take into account the current system resources and any special placement strategies. You can tune a worker node for a specific use case, like high connections, and then create rules to ensure that the applications that need that feature end up on that specific worker node. This is called *node affinity*. As a container orchestrator, you also need to restrict user authentication and authorizations. You can use an approach like role-based access control, which allows containers

to run in a safe and controlled manner. These approaches represent just a small part of the complex glue and wiring you'll need. It takes a whole framework to successfully deploy and manage containers.

Kubernetes Workload Resources

A *resource* is a type of object that encapsulates state and intent. To make this concept a little clearer, let's consider an automobile analogy. If a workload running on Kubernetes were a car, the resources would describe the parts of the car. For example, you could set your car to have two seats and four doors. You would not have to understand how to make a seat or a door. You would just need to know that Kubernetes will maintain the given count for both (no more, no less). Kubernetes resources are defined in a file called a *manifest*. Throughout this chapter, we will use the terms *resource* and *object* interchangeably.

NOTE *To learn more about resources and other concepts, visit* https://kubernetes.io/docs/concepts/.

Let's look at the most commonly used Kubernetes resources in a modern application stack.

Pods

Pods are the smallest building blocks in Kubernetes, and they form the foundation for everything interesting you'll do with containers. A Pod is made up of one or more containers that share network and storage resources. Each container can connect to the other containers, and all containers can share a directory between them by a mounted volume. You won't deploy Pods directly; instead, they'll be incorporated into a higher-level abstraction layer like a ReplicaSet.

ReplicaSet

A *ReplicaSet* resource is used to maintain a fixed number of identical Pods. If a Pod is killed or deleted, the ReplicaSet will create another Pod to take its place. You'll only want to use a ReplicaSet if you need to create a custom orchestration behavior. Typically, you should reach for a Deployment to manage your application instead.

Deployments

A *Deployment* is a resource that manages Pods and ReplicaSets. It is the most widely used resource for governing applications. A Deployment's main job is to maintain the state that is configured in its manifest. For example, you can define the number of Pods (which are called *replicas* in this context) along with the strategy for deploying new Pods. The Deployment resource controls a Pod's lifecycle—from creation, to updates, to scaling,

to deletion. You can also roll back to earlier versions of a Deployment if needed. Anytime your application needs to be long lived and fault tolerant, a Deployment should be your first choice.

StatefulSets

A *StatefulSet* is a resource for managing stateful applications, such as PostgreSQL, ElasticSearch, and etcd. Similar to a Deployment, it can manage the state of Pods defined in a manifest. However, it also adds features like managing unique Pod names, managing Pod creation, and ordering termination. Each Pod in a StatefulSet has its own state and data bound to it. If you are adding a stateful application to your cluster, choose a StatefulSet over a Deployment.

Services

Services allow you to expose applications running in a Pod or group of Pods within the Kubernetes cluster or over the internet. You can choose from the following basic Service types:

`ClusterIP` This is the default type when you create a Service. It is assigned an internal routable IP address that proxies connections to one or more Pods. You can access a `ClusterIP` only from within the Kubernetes cluster.

`Headless` This does not create a single-service IP address. It is not load balanced.

`NodePort` This exposes the Service on the node's IP addresses and port.

`LoadBalancer` This exposes the Service externally. It does this either by using a cloud provider's component, like AWS's Elastic Load Balancing (ELB), or a bare-metal solution, like MetalLB.

`ExternalName` This maps a Service to the contents of the `externalName` field to a `CNAME` record with its value.

You'll use `ClusterIP` and `LoadBalancer` the most. Note that only the `LoadBalancer` and `NodePort` Services can expose a Service outside the Kubernetes cluster.

Volumes

A *Volume* is basically a directory, or a file, that all containers in a Pod can access, with some caveats. Volumes provide a way for containers to share and store data between them. If a container in a Pod is killed, the Volume and its data will survive; if the entire Pod is killed, the Volume and its contents will be removed. Thus, if you need storage that is not linked to a Pod's lifecycle, use a *Persistent Volume (PV)* for your application. A PV is a resource in a cluster just like a node. Pods can use the PV resource, but the PV does not terminate when the Pod does. If your Kubernetes cluster is running in AWS, you can use *Amazon Elastic Block Storage (Amazon EBS)* as your PV. This makes Pod catastrophes easier to survive.

Secrets

Secrets are convenient resources for safely and reliably sharing sensitive information (such as passwords, tokens, SSH keys, and API keys) with Pods. You can access Secrets either via environment variables or as a Volume mount inside a Pod. Secrets are stored in a RAM-backed filesystem on the Kubernetes nodes until a Pod requests them. When not used by a Pod, they are stored in memory, instead of in a file on disk. However, be careful because the Secrets manifest expects the data to be in Base64 encoding, which is not a form of encryption.

With Secrets, sensitive information is kept separate from the application. This is because such information is more likely to be exposed in the continuous integration/continuous deployment process than if it's living in a resource. You still need to keep your Secret manifests safe by using RBAC to restrict broad access to the Secrets API. You can also store the sensitive data encrypted in the Secret and have another process to decrypt it on the Pod once it is mounted or needed. Another option is to encrypt the manifests locally before adding them to version control. No matter which method you choose, make sure you have a secure plan for storing sensitive information in Secrets.

ConfigMaps

ConfigMaps allow you to mount nonsensitive configuration files inside a container. A Pod's containers can access the ConfigMap from an environment variable, from command line arguments, or as a file in a Volume mount. If your application has a configuration file, putting it into a ConfigMap manifest provides two main benefits. First, you can update or deploy a new manifest file without having to redeploy your whole application. Second, if you have an application that watches for changes in a configuration file, then when it gets updated, your application will be able to reload the configuration without having to restart.

Namespaces

The *Namespace* resource allows you to divide a Kubernetes cluster into several smaller virtual clusters. When a Namespace is set, it provides a logical separation of resources, even though those resources can reside on the same nodes. If you don't specify a Namespace when creating a resource, it will inherit the Namespace cleverly named *default*. If your team has many users and a lot of projects spread among them, you might split those teams or applications into separate Namespaces. This makes it easy to apply secure permissions or other constraints to only those resources.

NOTE *This is* not *the same namespace you learned about in Chapter 6. That is a Linux kernel feature.*

Deploying the Sample telnet-server Application

To start exploring Kubernetes, you'll create a Deployment and two Services for the telnet-server application. I have chosen a Deployment to provide fault tolerance for your application. The two Services will expose the telnet-server application port and the application metrics port. By the end of this section, you'll have a Kubernetes Deployment with two Pods (replicas) running the telnet-server application that can be accessed from your local host.

Interacting with Kubernetes

Before you can deploy your telnet-server application, you'll need to make sure you can connect to your Kubernetes cluster. The most direct way to interact with the cluster is to use the `kubectl` command line application, which you can get in two ways. The first way is to download the standalone binary from *https://kubernetes.io/docs/tasks/tools/install-kubectl/* for your specific OS. The second way, which you'll use here, is to leverage minikube's built-in support for `kubectl`. Minikube will fetch the `kubectl` binary for you the first time you invoke the `minikube kubectl` command (if it's not already installed).

When using `minikube kubectl`, most commands will require double dashes (`--`) between `minikube kubectl` and subcommands. The standalone version of `kubectl`, however, does not need dashes between the commands. If you already have `kubectl` installed or want to use the standalone version, drop the `minikube` prefix and the double dashes from all the examples that follow.

Let's start out with a simple command so minikube can download the `kubectl` binary and test access to the cluster. Use the **`cluster-info`** subcommand for this example to verify that the cluster is up and running:

```
$ minikube kubectl cluster-info
Kubernetes master is running at https://192.168.99.109:8443
--snip--
```

You'll want to see similar output that indicates you can connect to the Kubernetes cluster. If there were an issue with talking to the cluster, you might see an error like `"The control plane node must be running for this command"`. If that happens, enter the **`minikube status`** command to make sure minikube is still up and running.

Reviewing the Manifests

Now that you have access to the cluster, review the provided manifests for the Deployment and Services. Kubernetes manifests are files designed to describe the desired state for applications and services. They manage resources like Deployments, Pods, and Secrets. These files can either be in JSON or YAML; we use the YAML format for this book, purely out of preference. The manifest files should be kept under source control. You'll usually find the files co-residing with the application they describe.

I have provided the manifest files to create the telnet-server Deployment and two Services. The files are located in the repository at *https://github.com/bradleyd/devops_for_the_desperate/*. Navigate to the *telnet-server/* directory and list the files in the *kubernetes/* subdirectory. There, you should find two files. The first file, *deployment.yaml*, creates a Kubernetes Deployment with two Pods of the telnet-server container image. The second file, *service.yaml*, creates two separate Services. The first Service creates a `LoadBalancer` so you can connect to the telnet-server from outside the Kubernetes cluster. The other Service creates a `ClusterIP`, which exposes the metrics endpoint from within the cluster. Don't worry about the metrics port for this chapter—we'll use it in Chapter 9 when discussing monitoring and metrics.

These manifest files can be quite verbose, so we'll focus on the basic structure each file contains. To describe a complex object, you'll need multiple fields, subfields, and values to define how a resource behaves. Because of this, it can be difficult to write a manifest from scratch. Among all these fields and values, there is a subset of required fields called *top-level fields*. These are common across all manifest files. Understanding top-level fields makes it easier to remember and parse a manifest file. The four top-level fields are as follows:

`apiVersion` This is a Kubernetes API version and group, like apps/v1. Kubernetes uses versioned APIs and groups to deliver different versions of features and support for resources.

`kind` This is the type of resource you want to create, such as a Deployment.

`metadata` This is where you set things like names, annotations, and labels.

`spec` This is where you set the desired behavior for the resource (kind).

Each of these top-level fields contains multiple subfields. The subfields contain information such as name, replica count, template, and container image. For example, `metadata` has `name` and `labels` subfields. The formats for the fields can be different for each Kubernetes resource. I won't describe every field, but I'll often use the `labels` subfield. *Labels* provide a way for the user to tag a resource with identifiable key values. For example, you could add a label to all resources that are in the `production environment`.

```
--snip--
metadata:
  labels:
    environment: production
--snip--
```

You can use these `labels` to narrow down search results and group similar applications together, as with a frontend website and its backend database counterpart. You'll use `labels` later, when you invoke the `minikube kubectl` command.

Listing all the different field structures in a manifest file would take up a lot of real estate. Instead, you can explore the documentation in two different places. The Kubernetes documentation at *https://kubernetes.io/docs/concepts/overview/working-with-objects/* describes all the resources and provides examples. The second place to explore, which is my favorite, is the `explain` subcommand for `kubectl`. The `explain` subcommand describes the fields associated with each resource type. You can use the dot (`.`) notation as a type field separator when searching for nested fields. For example, to learn more about a Deployment's `metadata labels` subfield, enter the following in a terminal:

```
$ minikube kubectl -- explain deployment.metadata.labels
KIND:     Deployment
VERSION:  apps/v1

FIELD:    labels <map[string]string>

DESCRIPTION:
     Map of string keys and values that can be used to organize and categorize
     (scope and select) objects. May match selectors of replication
     controllers and services. More info:
     http://kubernetes.io/docs/user-guide/labels
```

Notice how this example first searches for the resource type, then its top-level field, and then the subfield.

Examining the telnet-server Deployment

Now that you have an understanding of the building blocks of a manifest file, let's apply what you've learned to the telnet-server Deployment manifest. I've broken the *deployment.yaml* file into sections to make it easier to dissect. The first section at the top of the file has the `apiVersion`, `kind`, and `metadata` fields:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: telnet-server
  labels:
    app: telnet-server
--snip--
```

The type (`kind`) is `Deployment`, which uses the Kubernetes API group `apps` and API version `v1`. Under the `metadata` field, the Deployment `name` is set to `telnet-server`, and the `labels` are set to `app: telnet-server`. You'll use this label when searching for the telnet-server Deployment later on.

The next section of the file contains the parent `spec` field that describes the behavior and specification of the Deployment. The `spec` field contains a lot of subfields and values:

```
--snip--
spec:
  replicas: 2
  selector:
    matchLabels:
      app: telnet-server
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
--snip--
```

First, `spec` describes the `replicas` count for the Deployment; it's set to `2` to reflect the number of Pods you want to run. Inside the `selector` field, `matchLabels` locates the Pods that this Deployment will affect. The key value used in `matchLabels` must match the Pod's template labels (more on this later).

The `strategy` field describes how to replace the current running Pods with new ones during a rollout. This example uses a `RollingUpdate`, which will replace one Pod at a time as it goes. This is the default strategy for a Deployment. The other option for strategy, `Recreate`, kills the current running Pods before creating the new ones.

The `maxSurge` and `maxUnavailable` keys control the number of Pods created and terminated. Here, it's set to bring up an extra Pod during a rollout, which temporarily brings the Pod count to `replicas` + 1 (or three, in this case). Once the new Pod is up and running, one of the old Pods will be terminated. Then, the process repeats until all the new Pods are running and the old Pods are terminated. These settings will ensure that there is always a Pod to serve traffic during a Deployment. See *https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#strategy/* for more information about strategy.

The next part of the `spec` section is the `template` field. This field (along with its subfields) describes the Pods that this Deployment will create. The major subfields in this section are `metadata` and `spec`:

```
--snip--
template:
    metadata:
      labels:
        app: telnet-server
    spec:
      containers:
      - image: dftd/telnet-server:v1
        imagePullPolicy: IfNotPresent
        name: telnet-server
        resources:
          requests:
            cpu: 1m
            memory: 1Mi
          limits:
```

```
        cpu: 500m
        memory: 100Mi
    ports:
    - containerPort: 2323
      name: telnet
    - containerPort: 9000
      name: metrics
```

Here, the `app: telnet-server` key value is added for each Pod in the Deployment, using the `labels` subfield under `template` and `metadata`. The `app: telnet-server` label matches the key and value you used earlier in the `spec selector:` field. (You'll use this label again when searching for the Pods later.)

The `containers` field sets the container image for the first container in the Pod. In this case, it's set to the *dftd/telnet-server:v1* image you built in Chapter 6. This container `name` is `telnet-server`, just like the Deployment. Using the same `name` isn't a requirement; the `name` could be any string you choose so long as it is unique among the containers in the Pod.

The next subfield under `containers` is `resources`, which controls CPU and memory for a container. You can define `requests` and `limits` for each container individually. The `requests` are used for Kubernetes scheduling (orchestration) and overall application health. If a container needs a minimum of 2GB of memory and one CPU to start, you don't want Kubernetes to schedule that Pod (container) on a node that has only 1GB of memory or no CPUs available. `Requests` are the minimum resources your application needs. `Limits`, on the other hand, control the maximum CPU and memory a container can use on a node. You don't want a container to use all the memory or CPU on a node while starving any other containers running on it. In this example, the `CPU` limit is set to `500m` (millicpu), or half of a CPU. This unit can also be expressed as a decimal, like 0.5. In Kubernetes, one CPU is equivalent to one CPU core. The `memory` limit is set to `100Mi`, or 104,857,600 bytes. In Kubernetes, `memory` is expressed in bytes, but you can use more familiar units like M, Mi, G, and Gi. When these `limits` are set and the telnet-server container consumes more than `100Mi` of `memory`, Kubernetes will terminate it. However, if the `CPU` limit (`500m`) is surpassed, Kubernetes won't just kill the container. It will throttle, or limit, the CPU request time for that container. For more details on how Kubernetes quantifies resources, see *https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/*.

NOTE *Setting `requests` and `limits` for each resource you deploy in a k8s cluster will save you a lot of investigation time when things can't be scheduled or get killed. It will also save you money, since you'll use your nodes more efficiently.*

The container `ports` field sets the exposed ports you want to announce. This example exposes ports `2323` (`telnet`) and `9000` (`metrics`). These port definitions are for informational purposes only and have no bearing on whether a container can receive traffic. They simply let the user and Kubernetes know on what ports you expect the container to be listening.

Examining the telnet-server Service

The next manifest to examine is the `Service` resource. The *service.yaml* file creates two separate Services: one to expose the telnet-server and the other to expose the application metrics. We'll look at only the `telnet` Service and specific fields here since the `metric` Service is almost identical:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: telnet-server
  name: telnet-server
spec:
  ports:
  - port: 2323
    name: telnet
    protocol: TCP
    targetPort: 2323
  selector:
    app: telnet-server
  type: LoadBalancer
--snip--
```

A `Service` resource is set in the `kind` field, which is different from the Deployment manifest shown earlier. The Service `name` can be anything, but it must be unique within a Kubernetes Namespace. I've kept the `names` consistent with the rest of the resources here, for ease of use. I've also used the same `app: telnet-server` label to make finding things uniform and simple.

The `ports` field tells the `Service` which port to expose and how to connect it to the Pods. This exposes port `2323` (`telnet`) and forwards any traffic to port `2323` on the Pod.

Just as with the `selector` field for a Deployment, a `Service` uses a `selector` field to find the Pods to forward traffic to. This instance uses the familiar Pod label `app: telnet-server` as the match for the `selector`, which means any Pods with the label `app: telnet-server` will receive traffic from this `Service`. If there is more than one Pod, like in the Deployment, the traffic will be sent to all the Pods in a round-robin manner. Since the goal of the `telnet-server` application is to be exposed outside the cluster, it's set as a `LoadBalancer`.

Creating a Deployment and Services

It is time to create the Deployment and Services. To turn the sample application into a Kubernetes Deployment, you'll use the `minikube kubectl` command line tool and the manifest files you just reviewed (*https://github.com/bradleyd/devops_for_the_desperate/*).

To create and update resources, you can pass `minikube kubectl` two subcommands: `create` and `apply`. The `create` subcommand is *imperative*, which means it makes the resource reassemble the manifest file. It also throws an error if the resource already exists. The `apply` subcommand is *declarative*, which means it creates the resource if it does not exist and updates it if it

does. For this scenario, you'll use the `apply` command with an `-f` flag to instruct `kubectl` to run the operation against all the files in the *kubernetes/* directory. The `-f` flag can take filenames in lieu of directories as well.

From within the *telnet-server/* directory, enter the following command to create the `Deployment` and two `Services`:

```
$ minikube kubectl -- apply -f kubernetes/
deployment.apps/telnet-server created
service/telnet-server-metrics created
service/telnet-server created
```

The output should show that all three resources have been `created`. Be sure to investigate any errors if they arise from this command. Common errors you might see are usually due to syntax errors or typos in the YAML file.

Viewing the Deployment and Services

Once the telnet-server Deployment and Services are created, you need to know how to find them. Kubernetes provides multiple ways to view any object's status. The easiest method is to use the `minikube kubectl -- get <resource> <name>` command.

You can start by fetching the Deployment status by its name and then explore the Services. Enter the following to get the `Deployment` status for the `telnet-server`:

```
$ minikube kubectl -- get deployments.apps telnet-server
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
telnet-server   2/2     2            2           7s
```

The output should show that the `telnet-server` Deployment has two replicas (Pods) running (`2/2 READY`) and that they have been running for seven seconds (`7s AGE`). This should match the number of replicas set in the Deployment manifest. The `UP-TO-DATE` and `AVAILABLE` columns show how many Pods were updated to get to the desired number (`2`) and how many are available (`2`) to users, respectively. In this case, Kubernetes believes the Deployment is up and running and fully available.

You can also run the `minikube kubectl get pods` command to find out whether a Deployment is ready for traffic. Because you could have hundreds of Pods, you want to narrow down your results with the `-l` label filter flag. Enter the following to show only the `telnet-server` Pods:

```
$ minikube kubectl -- get pods -l app=telnet-server
NAME                             READY   STATUS    RESTARTS   AGE
telnet-server-775769766-2bmd5    1/1     Running   0          4m34s
telnet-server-775769766-k9kx9    1/1     Running   0          4m34s
```

This command lists any Pods that have the label `app: telnet-server` set; it's the same label set in the *deployment.yaml* file under the `spec.template.metadata.labels` field. The output shows two `telnet-server` Pods ready for traffic. You know this because the `READY` column shows `1/1` containers running and your

Deployment has only one container (`telnet-server`). If you had a Pod with multiple containers, you would want the number of running containers over the number of total containers to be the same.

NOTE *The `kubectl get <resource>` command is one you will use most often when interacting with Kubernetes.*

Now, use the same command as above but substitute `services` for the `pods` resource to display the two Services:

```
$ minikube kubectl -- get services -l app=telnet-server
NAME                   TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
telnet-server          LoadBalancer   10.105.187.105   <pending>     2323:30557/TCP   10m
telnet-server-metrics  ClusterIP      10.96.53.191     <none>        9000/TCP         10m
```

Since you used the same label (`app: telnet-server`) to organize your application, you can use the `-l` flag to find your match. The output shows that two Services were created about 10 minutes ago. One Service type is a `LoadBalancer`, and the other is a `ClusterIP`. The `LoadBalancer` is for exposing the `telnet-server` application. Don't be alarmed if your `EXTERNAL-IP` status is `<pending>`. Because you are running on minikube, no real `LoadBalancer` piece is included.

The `ClusterIP` Service allows the application metrics to be scraped from within the cluster. In this example, internal applications can reach the metrics endpoint by using either the `telnet-server-metrics` canonical name or the IP `10.96.53.191`. Using the canonical name is recommended.

Testing the Deployment and Services

Now that the telnet-server Deployment and Services are running, you'll want to test connectivity and availability. You want to be able to access the telnet-server application, like you did in Chapter 6, with the telnet client. After that, you'll test the Deployment's resiliency by killing a telnet-server Pod and watching it recover. Finally, you'll learn how to *scale*, meaning change the number of replicas that the Deployment has up and down from the command line, in the case of a change in load.

Accessing the Telnet Server

You'll use the `minikube tunnel` command to expose your `LoadBalancer` Service outside the Kubernetes cluster. This command will provide you with an IP address that you can use to connect, using the `telnet client` command again. The `tunnel` subcommand runs in the foreground, so it should be run in a terminal that won't get closed. The command also requires *root* privileges. If you do not have *root* privileges on your local machine, use the `minikube service` command instead. Visit *https://minikube.sigs.k8s.io/docs/commands/service/* for more details.

In a terminal, enter the following to create the network `tunnel` to the `telnet-server` Service:

```
$ minikube tunnel
Password:
Status:
        machine: minikube
        pid: 42612
        route: 10.96.0.0/12 -> 192.168.99.103
        minikube: Running
        services: [telnet-server]
    errors:
                minikube: no errors
                router: no errors
                loadbalancer emulator: no errors
```

After entering your password, the command outputs a `route`, the `services` exposed, and any present `errors`. Make sure you leave this running while you try to connect to the `telnet-server`. Once the `tunnel` is closed, all the connections will drop. Since there are `no errors` to report, the `tunnel` should be operational at this point. Don't do it now, but when you want to close the `tunnel`, press CTRL-C to shut it down.

Now, with the tunnel up, you need to get the new external IP address for the `LoadBalancer` Service. As a shortcut, pass the Service name to **`get services telnet-server`** (in this case) to view only the Service you are interested in:

```
$ minikube kubectl -- get services telnet-server
NAME            TYPE           CLUSTER-IP       EXTERNAL-IP      PORT(S)          AGE
telnet-server   LoadBalancer   10.105.187.105   10.105.187.105   2323:30557/TCP   15m
```

The `EXTERNAL-IP` column should now be populated with an IP address instead of `<pending>`. Here, the `telnet-server` application IP address is set to `10.105.187.105`, and the external `PORT` is set to `2323`. Your `EXTERNAL-IP` may be different from mine, so just use the IP from this column.

In another terminal that is not running the tunnel, use the `telnet client` command again (`telnet 10.105.187.105`) with the new IP address to access the `telnet-server`, as shown in Figure 7-2.

As you can see, the telnet-server responded with the ASCII art logo. Press Q to quit, since you are just testing connectivity. The `tunnel` command made it possible to hit the Service using an assigned IP like it was a public-facing application. If this were on a cloud provider like AWS, the IP would be accessible to anyone on the internet. Feel free to kill the `tunnel` command in the other terminal, but you'll use it again in future chapters.

Figure 7-2: Testing telnet access to telnet-server

Troubleshooting Tips

If you cannot connect to the telnet-server like in Figure 7-2, check that the Pods are still running and that they are reporting that `1/1` containers are `READY`. If the `READY` column shows `0/1` instead and the `STATUS` column has an error like `ImagePullBackOff` or `ErrImagePull`, then the Pod could not find the telnet-server image you built in Chapter 6. Make sure the image is built and available when you list the Docker images.

If the `READY` and `STATUS` columns are correct, the next step is to make sure your Service is wired up to your Pods. One way to check this connection is with the `kubectl get endpoints` command, which will tell you if the Service can find the Pods you specified in the Service `spec.selector` field located in the *service.yaml* file:

```
$ minikube kubectl -- get endpoints -l app=telnet-server
NAME                    ENDPOINTS                           AGE
telnet-server           172.17.0.3:2323,172.17.0.5:2323     20m
telnet-server-metrics   172.17.0.3:9000,172.17.0.5:9000     20m
```

The `ENDPOINTS` column shows the internal Pod IP addresses with ports. Since you have two Pods, there are two IP addresses separated by a comma for each Service. If the Service can't locate the Pods, the `ENDPOINTS` column will be set to `<none>`. If your `ENDPOINTS` column has `<none>`, check that the `spec.selector` field in your Service matches what is in the `spec.template.metadata.labels` field in the *deployment.yaml* file. I have preset it to the label `app: telnet-server` in the example. Having mismatched labels between a Service and a resource is a common mistake; it will happen to you at least once.

NOTE *Visit* https://kubernetes.io/docs/tasks/debug-application-cluster/debug-service/ *for more debugging tips and possible solutions.*

Killing a Pod

Another great feature of Deployments is recovery. Failure is going to happen, so embrace it! A Deployment will get you back up and to full strength in no time. Remember, a Deployment's main purpose is to keep the desired number of Pods running. To test this, you'll delete one of the `telnet-server` Pods and then watch the Deployment respawn another in its place. First, you'll need to fetch one of the `telnet-server` Pods' names and delete it.

Enter the following to get the `telnet-server` Pods again:

```
$ minikube kubectl -- get pods -l app=telnet-server
NAME                              READY   STATUS    RESTARTS   AGE
telnet-server-775769766-2bmd5   1/1     Running   0          25m
telnet-server-775769766-k9kx9   1/1     Running   0          25m
```

It really doesn't matter which Pod you delete, so just choose the first one on the list, which is `telnet-server-775769766-2bmd5` on my cluster. (Your Pod names will be different, as they are autogenerated.)

Now, enter the following command to `delete` the selected Pod:

```
$ minikube kubectl -- delete pod <telnet-server-775769766-2bmd5>
pod "telnet-server-775769766-2bmd5" deleted
```

The command might appear to hang for a few seconds, but it will eventually finish when the Pod has terminated.

If you list the Pods again, you'll see two Pods are still running, but now the `telnet-server-775769766-2bmd5` is gone and has been replaced with a new Pod:

```
$ minikube kubectl -- get pods -l app=telnet-server
NAME                              READY   STATUS    RESTARTS   AGE
telnet-server-775769766-k9kx9   1/1     Running   0          25m
telnet-server-775769766-rdg5w   1/1     Running   0          1m16s
```

This new Pod, named `telnet-server-775769766-rdg5w`, is more than a minute old, is `Running`, and is ready to accept connections.

Scaling

Let's pretend the telnet-server application really resonates with the nostalgic over-35 crowd and becomes a runaway success. The two telnet-servers will no longer be adequate for handling the increased traffic, so you'll need to scale up your replicas to a count greater than two. You can do this in two ways. The first way is to edit the *deployment.yaml* manifest file and apply the changes to the cluster using the `minikube apply` command. The second way is to use the `minikube kubectl scale` command. I'll demonstrate this example using the `minikube kubectl scale` command, since you already learned how to apply manifest files earlier in this chapter.

You are going to increase the Deployment replica count by one, bringing the total number of Pods to three. (In a real production environment,

you would base the replica count number off some key metrics instead of a finger in the wind.) Enter the following command to scale up the `telnet-server` Deployment:

```
$ minikube kubectl -- scale deployment telnet-server --replicas=3
deployment.apps/telnet-server scaled
```

The `scale deployment` command takes a `--replicas` flag to set the number of Pod replicas. The output shows the `telnet-server` Deployment has `scaled`, but let's verify this.

Enter the following command to verify that the replica count has changed for your Deployment:

```
$ minikube kubectl -- get deployments.apps telnet-server
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
telnet-server   3/3     3            3           17m
```

Here, you get the Deployment resource information for `telnet-server`. The Deployment has three out of three (`3/3`) replicas `READY`, up from the two it had earlier.

The `scale` command changes the replica count in real time on the cluster. This can be dangerous. If a colleague pushes out a new version of the `telnet-server` application right after you scaled from the command line, the replica state will not match. This is because when he or she runs the `minikube kubectl -- apply -f kubernetes/deployment.yaml` command, the Deployment replica count would go back to two, since that's what's stated in the *deployment.yaml* manifest file.

NOTE *Changing the replica count or editing any resources live via the command line is more trouble than it's worth, as doing so often causes a split brain between the running state and the saved state in your manifests. To save yourself time debugging and avoid causing your customers pain, always opt for infrastructure changes that are tracked and versioned in source control instead of quick-and-dirty live changes.*

Logs

The last piece of orchestration to test is accessing the telnet-server application logs. Fortunately, Kubernetes makes this simple with the `kubectl logs` subcommand. You want to grab the logs for all three of your telnet-server Pods. One way to do this is to execute the `logs` command for each of the three Pods and view the results. Enter the following command to view one of the Pods logs (remember, your Pod names will be different from mine):

```
$ minikube kubectl -- logs <telnet-server-775769766-rdg5w>
--snip--
```

This works fine if you do not have many Pods or if you know which Pod an event happened on. If not, a better option is to grab all the Pods logs

at the same time and mark each log line with the Pod name from which it came. Enter the following command to fetch all the logs for each Pod:

```
$ minikube kubectl -- logs -l app=telnet-server --all-containers=true --prefix=true
[pod/telnet-server-775769766-k9kx9/telnet-server] telnet-server: 2022/02/03 21:07:30 telnet-
server listening on [::]:2323
[pod/telnet-server-775769766-k9kx9/telnet-server] telnet-server: 2022/02/03 21:07:30 Metrics
endpoint listening on :9000
--snip--
```

Quite a few flags are used in this command; let's break each one down:

- To fetch only Pods with this label: `-l app=telnet-server`
- When you have multiple Pods and want to see all the logs: `--all-containers=true`
- Each log line with the Pod name from which the log came: `--prefix=true`

The output should show at least six log lines—two start-up log lines for each Pod (3) and whatever other logs may have shown up from connecting earlier with the `telnet` command. The log output is not important now, as you just need to make sure you can access the logs for your application.

Summary

In this chapter, you learned how to run the telnet-server container image inside a Kubernetes cluster. You successfully orchestrated your application by using a Kubernetes Deployment resource that you exposed to your local host via a Kubernetes Service. Finally, you explored how to create, query, and view your resources and logs with the `minikube kubectl` command. In the next chapter, you'll learn to automate the deployment of telnet-server by implementing a simple delivery pipeline inside Kubernetes.

8

DEPLOYING CODE

You have been methodically building up your infrastructure to get to this point, and you have put in place all the foundational pieces you need to run your application. You have built and deployed in the Kubernetes cluster the container image for the telnet-server application. If you want to release a new version of your application, all you need to do is rebuild the container image and then redeploy the Kubernetes manifests.

However, there are some glaring flaws within your setup. For one, you are not running any tests to verify that the code or container image is defect-free. Also, the way you have set it up, every time any code or configuration changes, you'll need to build the container image and release the Deployment manually. This manual process is fine for kicking the tires on

new technologies, but hopefully you have learned (and agree) that these steps can and should be automated. Successful software engineering teams often release small code changes using automation, allowing them to find errors quickly and reduce complexities in their infrastructure. As mentioned in an earlier chapter, this process of getting code from your editor to your stakeholders in a consistent and automated manner is usually referred to as *continuous integration and continuous deployment (CI/CD)*.

In this chapter, you're going to build a simple CI/CD pipeline for the telnet-server application using freely available tools. This pipeline will watch the telnet-server source code changes, and if there are any, it will kick off a series of steps to get the changes deployed to the Kubernetes cluster. By the end of this chapter, you'll have a local development pipeline that builds, tests, and deploys your code to the Kubernetes cluster using automation.

CI/CD in Modern Application Stacks

Continuous integration and continuous deployment are software development methodologies that describe the way code is built, tested, and delivered. The CI steps cover the testing and building of code and configuration changes, while the CD steps automate the deployment (or delivery) of new code.

During the CI stage, a software engineer introduces new features or bug fixes through a version control system like Git. This code gets run through a series of builds and tests before finally producing an artifact like a container image. This process solves the "works on my machine" problem because everything is tested and built in the same way to produce a consistent product. The testing steps usually consist of unit tests, integration tests, and security scans. The unit and integration tests make sure the application behaves in an expected manner, whether in isolation or interacting with other components in your stack. The security scans usually check for known vulnerabilities in your applications software dependencies or for vulnerable base container images you are importing. After the testing steps, the new artifact is built and pushed to a shared repository, where the CD stage has access to it.

During the CD stage, an artifact is taken from a repository and then deployed, usually to production infrastructure. CDs can use different strategies to release code. These strategies are usually either *canary*, *rolling* (in our case), or *blue-green*. See Table 8-1 for more information on each strategy.

The idea behind deployment strategies is to minimize problematic code before it can have an impact on many users. The infrastructure you'll be deploying to most likely will be a container orchestrator like our Kubernetes cluster, but it could just as easily be VMs in a cloud provider.

Table 8-1: Deployment Strategies

Name	Description
Canary	This strategy rolls out new code so only a small subset of users can access it. If the canary's code presents zero errors, the new code can be rolled out further to more customers.
Blue-Green	In this strategy, a production service (blue) takes traffic while the new service (green) is tested. If the green code is operating as expected, the green service will replace the blue service, and all customer requests will funnel through it.
Rolling	This strategy deploys new codes one by one, alongside the current code in production, until it is fully released.

After the deployment is successful, a monitoring step should observe the new code and make sure nothing has slipped past the CI phase. If a problem is detected, like high latency or increased error counts, it will be no problem to roll back the application to a previous version that was deemed safe. This is one of the great features of a container orchestrator like Kubernetes. It makes rolling code forward and backward very simple. (We'll test the rollback feature later.)

Setting Up Your Pipeline

Before creating your pipeline, you'll need to install a few tools to help automate code building, testing, and delivery. There are many tools on the market that do this, but for our scope, I am using two pieces of software that are open source and integrate nicely with Kubernetes. The first tool is called Skaffold, and it helps with continuous development for Kubernetes-native applications. It will make setting up the CI/CD pipeline to the local k8s cluster easy. If Skaffold is not installed, follow the instructions at *https://skaffold.dev/docs/install/* for your OS to complete the installation.

The other tool, `container-structure-test`, is a command line application that validates the container image's structure after it's built. It can test whether the image was constructed properly by verifying whether a specific file exists, or it can execute a command and validate its output. You can also use it to verify that a container image was built with the correct metadata, like the ports or environment variables you would set in a Dockerfile. The installation instructions for `container-structure-test` are available at *https://github.com/GoogleContainerTools/container-structure-test/*.

NOTE *Both tools are ever changing and may not be considered production worthy by the time you read this. The main goal of this section is to show you how the pipeline process works and how you can create it with little effort on your local machine.*

Reviewing the skaffold.yaml File

The *skaffold.yaml* file describes how to build, test, and deploy your application. This file should live in the root of your project and be kept under version control. The YAML file has many different options to choose from, but your pipeline will focus on three main sections: `build`, `test`, and `deploy`. The `build` section describes how to build your container image, the `test` section describes what tests to perform, and the `deploy` section describes how to release your application to the Kubernetes cluster.

The *skaffold.yaml* file is in the *telnet-server/* directory inside the cloned repository (*https://github.com/bradleyd/devops_for_the_desperate/*). You don't need to edit or open this file, but you should have some familiarity with its basics and structure.

```
--snip--
kind: Config
build:
  local: {}
  artifacts:
  - image: dftd/telnet-server
test:
- image: dftd/telnet-server
  custom:
  - command: go test ./... -v
  structureTests:
  - ./container-tests/command-and-metadata-test.yaml
deploy:
  kubectl:
    manifests:
    - kubernetes/*
```

The `build` section uses the default build action, which is the `docker build` command, to create our container image locally. The container `image` name is set to `dftd/telnet-server`. This matches the same image name you are using in the *deployment.yaml* file. You'll see why that is important when you look at the `deploy` section. The Skaffold tool precalculates the container image tag using the current Git commit hash, which is the default behavior. The generated tag is appended to the container image name automatically, and it's conveniently set to an environment variable (`$IMAGE`) that can be referenced if needed.

NOTE *A Git commit hash is a unique ID that Git uses to mark the repository state at a particular point in time.*

The `test` section allows you to run any tests against the application and container image. In this case, you'll use unit tests that exist for the `telnet-server` application that I've provided for you. The unit tests, which are under the `custom` field, run the `go test` command for all the test files. This step requires that the Go programming language be installed. If you do not have Go installed, follow the instructions at *https://go.dev/doc/install/* for your OS.

The next test that gets run is `structureTests`. This test checks the final container image for defects. We'll go over these container tests briefly in a later section.

Finally, the `deploy` section uses the Kubernetes manifest files inside the *kubernetes/* directory to release the `telnet-server` Deployment. The Skaffold tool performs a patch against the running Deployment and replaces the current container image and tag (which is *dftd/telnet-server:v1*) with the new one Skaffold generated during the `build` step. Because these names match the tag, they can be easily updated to a new one in the pipeline.

Reviewing the Container Tests

Once the telnet-server container image is built and the application tests pass, the container tests are run on the newly built image. The container tests are located in a subdirectory called *container-tests/*, which is under the *telnet-server/* directory. This directory contains one test file named *command-and-metadata-test.yaml*. In this file, I have provided one application test to make sure the binary was built correctly, and I have also provided a few container image tests to verify that the container was built with the expected instructions.

You should review the structure tests now. Open the YAML file in your editor or follow along below:

```
--snip--
commandTests:
  - name: "telnet-server"
    command: "./telnet-server"
    args: ["-i"]
    expectedOutput: ["telnet port :2323\nMetrics Port: :9000"]
metadataTest:
  envVars:
    - key: TELNET_PORT
      value: 2323
    - key: METRIC_PORT
      value: 9000
  cmd: ["./telnet-server"]
  workdir: "/app
```

The `commandTests` command executes the `telnet-server` binary, passing the `-i` (info) flag to it to output the ports on which the application is listening to STDOUT. The command output is then matched against what is in the `expectedOutput` field. For a successful test, the output should match `telnet port :2323\nMetrics Port: :9000` so you can make sure your binary was compiled correctly during the container `build` phase. This test makes sure the `telnet-server` application can at least run and function on a basic level.

The `metadataTest` looks to see whether the container image was built with the proper instructions in the Dockerfile. The metadata tests verify environment variables (`envVars`), command (`cmd`), and `workdir`. These tests are useful for catching any delta between Dockerfile changes across different commits.

Simulating a Development Pipeline

Now that you understand the pipeline configuration, let's get a running pipeline. You can execute the `skaffold` command with either the `run` or the `dev` subcommand. The `run` subcommand is a one-off that builds, tests, and deploys the application and then exits. It does not watch for any new code changes. The `dev` command does everything `run` does, but it watches the source files for any changes. Once it detects a change, it kicks off the `build`, `test`, and `deploy` steps described in the *skaffold.yaml* file. For this example, you'll use the `dev` subcommand to simulate a development pipeline.

After the `dev` subcommand is run successfully, it will wait and block looking for any changes. By default, you'll need to press CTRL-C to exit the `skaffold dev` mode. However, when you use CTRL-C to exit, the default behavior is to clean up after itself by removing the telnet-server Deployment and Services from the Kubernetes cluster. Since you'll be using the telnet-server Deployment throughout this chapter and book, add the `--cleanup=false` flag to the end of the `dev` command to bypass this behavior. This way, the Pods will stay running after you quit the command.

To kick off the pipeline, make sure you are in the *telnet-server/* directory and your Kubernetes cluster is still running. The `skaffold` command can be quite chatty when executed. To make it easier to follow, you'll break down the output as it aligns with the three `skaffold` sections above (`build`, `test`, and `deploy`).

Enter the following command in a terminal to run `skaffold`:

```
$ skaffold dev --cleanup=false
Listing files to watch...
 - dftd/telnet-server
Generating tags...
 - dftd/telnet-server -> dftd/telnet-server:4622725
Checking cache...
 - dftd/telnet-server: Not found. Building
Found [minikube] context, using local docker daemon.
Building [dftd/telnet-server]...
--snip--
Successfully tagged dftd/telnet-server:4622725
```

The first action this command executes is to set the container tag to `4622725`, after which the Docker image is built. Your tag will likely be different, as it's based off the current Git commit hash of my repository.

After a successful build, `skaffold` triggers the test section where the unit and container infrastructure tests are kept:

```
Starting test...
Testing images...
===========================================================
====== Test file: command-and-metadata-test.yaml ======
===========================================================
=== RUN: Command Test: telnet-server
```

```
--- PASS
duration: 571.602755ms
stdout: telnet port :2323
Metrics Port: :9000

=== RUN: Metadata Test
--- PASS
duration: 0s

========================================================
====================== RESULTS =======================
========================================================
Passes:      2
Failures:    0
Duration:    571.602755ms
Total tests: 2

PASS
Running custom test command: "go test ./... -v"
?       telnet-server    [no test files]
?       telnet-server/metrics    [no test files]
=== RUN   TestServerRun
Mocked charge notification function
    TestServerRun: server_test.go:23: PASS:        Run()
--- PASS: TestServerRun (0.00s)
PASS
ok      telnet-server/telnet     (cached)
Command finished successfully.
```

The container tests and `telnet-server` unit tests pass with zero errors.

Finally, after the container is built and all the tests pass, `skaffold` attempts to deploy the container to Kubernetes:

```
--snip--
Starting deploy...
 - deployment.apps/telnet-server created
 - service/telnet-server created
 - service/telnet-server-metrics created
Waiting for deployments to stabilize...
 - deployment/telnet-server: waiting for rollout to finish: 0 of 2 updated
replicas are available...
    - pod/telnet-server-6497d64d7f-j8jq5: creating container telnet-server
    - pod/telnet-server-6497d64d7f-sx5ll: creating container telnet-server
 - deployment/telnet-server: waiting for rollout to finish: 1 of 2 updated
replicas are available...
 - deployment/telnet-server is ready.
Deployments stabilized in 2.140948622s
Press Ctrl+C to exit
Watching for changes...
```

The Deployment is using our Kubernetes manifest files for the `telnet-server` application. For this Deployment, `skaffold` is using the new container image and tag (*dftd/telnet-server:4622725*) that was just built and tested to

replace the one that is currently running (*dftd/telnet-server:v1*). If the `build`, `test`, and `deploy` steps are successful, there will not be any visible errors, and the final line should say, "`Watching for changes`." If there are errors in any of the steps, the pipeline will halt immediately and throw an `error` with some clues to where the fault occurred. If any errors do occur, tack the `--verbosity debug` flag onto the `skaffold dev` command to increase the output's verbosity.

If the container image and tag already exist, `skaffold` will skip the `build` and `test` sections and go right to the `deploy` step. This is a great time-saver, as you won't need to repeat all the steps if all you are doing is redeploying the same container image. If your repository has uncommitted changes, `skaffold` adds `-dirty` to the end of your tag (`4622725-dirty`) to signal that changes are yet to be committed. In most cases, you'll see this often when developing locally. That is because you'll likely be constantly tinkering and making changes before committing your code.

Making a Code Change

The pipeline is now set up, so you'll want to make a code change to test the workflow. Let's try something simple, like changing the color of the DFTD banner that greets you when you connect to the telnet-server. The source code for telnet-server is located in the *telnet-server/* directory. Currently, the banner is set to green (my favorite color). Once you make the code change and save the file, `skaffold` should recognize the change and trigger `build`, `test`, and `deploy` again.

In a different terminal from the one in which you are already running `skaffold`, open the *banner.go* file, located in the *telnet/* subdirectory, using your favorite editor. Don't worry about the code or the file's contents; you're just going to change the color. On line 26, you'll see some code that looks like this:

```
return fmt.Sprintf("%s%s%s", colorGreen, b, colorReset)
```

This is the line that sets the banner color.

Replace the string `colorGreen` with the string `colorYellow`, so the line now looks like this:

```
return fmt.Sprintf("%s%s%s", colorYellow, b, colorReset)
```

After the change, save and close the file. Head back to the terminal where you are running the `skaffold dev` command. You should now see new activity that looks very similar to the output from the first `skaffold` run. All the steps will have been triggered again because you made a change in the source code that `skaffold` watches. The end result should be the same: you will have completed the Deployment rollout, and two new Pods will be running. If that isn't the case, make sure that you actually saved the *banner.go* file and that `skaffold dev` is still running.

Testing the Code Change

Next, you should make sure the new code was delivered to the Kubernetes cluster. Do this by validating that the DFTD banner color changed from green to yellow.

In the previous chapter, you used the `minikube tunnel` command to access the telnet-server application. If you still have it running in a terminal, jump to the telnet client instructions below. If not, open another terminal and run the **`minikube tunnel`** command once again.

You'll need the IP address of the telnet-server Service again to access it. Run this command to get the telnet-server Service IP:

```
$ minikube kubectl -- get services telnet-server
NAME           TYPE           CLUSTER-IP       EXTERNAL-IP        PORT(S)       AGE
telnet-server  LoadBalancer   10.105.161.160   10.105.161.160 2323:30488/TCP   6m40s
```

Your `EXTERNAL-IP` may be different from mine, so use the IP from that column and port `2323`.

Access the application again with the **`telnet`** client command, as follows:

```
$ telnet 10.105.161.160 2323
```

The DFTD banner, shown in Figure 8-1, should now be yellow.

Figure 8-1: The telnet session should have a yellow banner

If it's not yellow, go back and make sure that the color was changed in the code correctly and that the file was saved. Also, you can use the `minikube kubectl get pods` command to verify that you have new Pods running. Make sure the age of the Pods goes back to within a short time after you saved the *banner.go* file. You should also look at the output in the terminal where `skaffold dev` is running, to detect any noticeable errors.

Testing a Rollback

There will be times when you need to roll back the application you have deployed. This can be due to many reasons, from problematic code to misalignment between product and engineering. Let's say you wanted to go back to the release where the welcome banner was green. You would have two choices. On the one hand, you could make the necessary code change to set the banner back to green and put the application back through the CI/CD pipeline again. On the other hand, you could roll back the Deployment to the older version, where the DFTD banner is green. We'll explore the latter option.

If the troubled application does not pose any immediate service disruption or cause ongoing customer impacts, you should make a hotfix for the code and follow your release cycle through your CI/CD pipeline. But what if this bug (error) caused a service disruption to your customers as soon as you deployed the code? You might not have time to wait for a thorough investigation to happen and a hotfix to run through the pipeline. But Kubernetes provides a way to roll back a Deployment, and other resources, to a previous revision. So in this case, you'll roll back only one revision, to when the banner was green.

First, check the rollout history. Every time you deploy new code, Kubernetes tracks the Deployments and saves the resource state at that given time. Enter the following in a terminal to fetch the Deployment history for `telnet-server`:

```
$ minikube kubectl -- rollout history deployment telnet-server
deployment.apps/telnet-server
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

If you have been following along without any hiccups, the output should show two tracked Deployments. Currently, `REVISION 2` is active. Notice the `CHANGE-CAUSE` column has `<none>`. That is because you did not tell Kubernetes to record the change. Using the `--record` flag when running `kubectl apply` makes Kubernetes record which command triggered the `deploy`. Don't worry about using `--record` for this book. Depending on how many times you deployed the manifests from Chapter 7 or how many times you ran `skaffold dev`, your `REVISION` numbers may be different. The actual number doesn't matter here; you're just going back to the previous revision.

Let's force a rollback from the command line to `REVISION 1`, which should reapply the manifests used in the first `deploy`, when the banner was green. The `kubectl rollout` command has an `undo` subcommand for this case:

```
$ minikube kubectl -- rollout undo deployment telnet-server --to-revision=1
deployment.apps/telnet-server rolled back
```

You can leave off the `--to-revision=1` flag, as the default is to roll back to the previous revision. I added it here in case you ever need to roll back to a revision that was not the previous one.

In a few seconds, the previous release should be running and accepting new connections. Verify this by running the **`minikube kubectl get pods`** command to show the Pods are new and have been running for only a few seconds:

```
$ minikube kubectl -- get pods
NAME                              READY   STATUS    RESTARTS   AGE
telnet-server-7fb57bd65f-qc8rg    1/1     Running   0          28s
telnet-server-7fb57bd65f-wv4t9    1/1     Running   0          29s
```

These Pods' names have changed, and the Pods have been running for only 29 seconds, which is what you'd expect after just rolling them back.

Now, check the banner's color. Make sure the `minikube tunnel` command is still running, and then enter the **`telnet`** command into the application one more time:

```
$ telnet 10.105.161.160 2323
```

If everything went well, your DFTD banner should be green again.

If you run the `rollout history` command again, the current revision deployed will be `3`, and the previous revision, when the banner was yellow, will be `2`.

You now know how to do an emergency rollback in Kubernetes, to recover from any immediate service disruption. This technique can be useful when your organization focuses on *mean time to recovery (MTTR)*, which basically means how long it takes for a service to go from "down" to "up" from a customer's point of view.

Other CI/CD Tooling

Development pipelines are complex pieces of your infrastructure. In my quest to break them down in a simple manner, I've oversimplified some aspects. However, my main goal has been to show you how to create a simple pipeline to test and deploy code on a local Kubernetes cluster. You can also use this same pattern in nonlocal setups, like the ones in AWS or Google. The common strands that bind these processes together are portability and the use of a single file to describe the pipeline for an application. This means that if your pipeline YAML file works locally, it should also work on remote infrastructure.

That said, it might be helpful to describe some tools that are popular in the CI/CD space. There are more tools available that I can count, but popular ones include Jenkins, ArgoCD, and GitLab CI/CD. Of these, Jenkins is probably the most widely used, and it can operate both CI and CD for VMs, containers, and any other artifact you're using. There are also a lot of widely available community plug-ins that make Jenkins extensible, but a lot of security issues come with them. Be diligent about updating plug-ins and looking out for issues.

Jenkins can deploy to any infrastructure and use any version control for code repositories. Argo CD, on the other hand, is a Kubernetes deployment

tool that focuses only on the `deploy` phase. It can do canary or blue-green deployments out of the box, and it comes with a nice command line tool to manage the infrastructure. You can hook Argo CD into your pipeline after CI is done. Finally, GitLab CI/CD offers a full-featured pipeline (like Jenkins) that leverages Gitlab's version control product to manage code repositories. It was designed for DevOps and includes almost everything you need to get up and running in a modern infrastructure stack.

Although these tools do a good job of empowering you to have a pipeline, it is important to separate the philosophy behind CI/CD from the tools used in this space. The truth is, each organization you work at may or may not use the tools or processes described here. The methodologies, rather than the individual tools themselves, are what's important. No matter what tools you use, the main goal behind CI/CD is to validate and deliver code in small, predictable iterations, thus reducing the chance of errors or defects.

Summary

This chapter introduced you to continuous integration and continuous deployment methodologies. The CI/CD pipeline you created used two tools to `build`, `test`, and `deploy` code. This allowed you to automate an application's lifecycle in a Kubernetes cluster. You also learned about a rollback feature built into Kubernetes that makes it easy to recover quickly from errant code or misconfigured releases.

This concludes Part II, which has focused on containerization and orchestration. You now can build and deploy a simple application inside a Kubernetes cluster. Going forward, we'll shift gears and discuss observability, with a focus on metrics, monitoring, and alerting. We'll also explore common troubleshooting scenarios you will find on a host or network, plus the tools you can use to diagnose them.

PART III

OBSERVABILITY AND TROUBLESHOOTING

9

OBSERVABILITY

Observability is an attribute of a system, rather than something you do. It is a system's ability to be monitored, tracked, and analyzed. Any application worthy of production should be observable. Your main goal in observing a system is to discern what it is doing, internally. You do this by analyzing system outputs like metrics, traces, and logs. *Metrics* usually consist of data over time that provide key insights into an application's health and/or performance. *Traces* track a request as it traverses different services, to provide a holistic view. *Logs* provide a historical audit trail of errors or events that can be used for troubleshooting. Once you collect this data, you need to monitor it and alert someone when there is unexpected behavior.

It is not necessary to analyze metrics, traces, and logs from every application or piece of architecture. For example, tracing is key when you are running distributed microservices because it can shed light on the individual state of a given service and its interactions with other services. Your decisions about what, how, and how much to observe really will hinge on

the level of architectural complexity you are dealing with. Since your application and infrastructure are relatively uncomplicated, you'll observe your telnet-server application with metrics, monitoring, and alerting.

In this chapter, you'll first install a monitoring stack inside the Kubernetes cluster you created in Chapter 7. Then, you'll investigate common metric patterns you can use as a starting point for any service or application you may encounter. Finally, you'll configure the monitoring stack to send an email notification when an alert is triggered. By the end of this chapter, you'll have a solid understanding of how to install, monitor, and send notifications for any application inside Kubernetes.

Monitoring Overview

Monitoring is any action that entails recording, analyzing, and alerting on predefined metrics to understand the current state of a system. To measure a system's state, you need applications to publish metrics that can tell a story about what the system is doing at any given time. By setting thresholds around metrics, you can create a baseline of what you expect the application's behavior to be. For example, a web application is expected to respond with an HTTP 200 in most cases. When the application's baseline is not in a range you expect, you'll need to alert someone so they can bring the application back into line. Systems will fail, but robust monitoring and alerting can be the bridge to user satisfaction and on-call shifts that end with you getting a good night's sleep.

An observable system should do its best to answer two main questions: "What?" and "Why?" "What?" asks about a symptom of an application or service during a specific time frame, while "Why?" asks for the reasons behind the symptom. You can usually get the answer to "What?" by monitoring symptoms, while you can get the answer to "Why?" by other means, like logs and traces. Correlating the symptom with the cause can be the hardest part of monitoring and observability. This means your application's resiliency is only as good as the data the application outputs. A common phrase to describe this concept is "Garbage in, garbage out." If the metrics exported from an application are not targeted or relevant to how the user interacts with the service, detecting and diagnosing issues will be more difficult. Because of this, it's more important to measure the application's *critical path*, or its most-used parts, than every possible use case.

For instance, say you go to your doctor because you woke up with nausea and stomach cramps. The doctor asks you some basic questions and takes your temperature, heart rate, and blood pressure. While your temperature is a bit elevated, everything else falls within the normal range. After reviewing all the data, the doctor makes a judgment call about why you feel bad. Odds are, the doctor will be able to correctly diagnose your ailment (or at least find more clues about it to follow up on).

This process of medical diagnosis is the same process you'll follow when diagnosing application issues. You'll measure the symptoms and try to explain them with a diagnosis or a hypothesis. If you have enough relevant data points, it will be easier for you to correlate the symptoms with a cause.

In the example above, if the doctor asked what you had eaten recently (another solid data point), they might have correlated your nausea and cramps with your unwise choice to eat gas station sushi at 3 AM.

Finally, always consider the "What?" and "Why?" when designing metrics and monitoring solutions for your applications. Avoid metrics or alerts that do not provide value to your stakeholders. Engineers who get bombarded by nonactionable alerts tend to get tired and ignore them.

Monitoring the Sample Application

You'll begin by monitoring the metrics that this book's example telnet-server publishes. The telnet-server application has an HTTP endpoint that serves up metrics about the application. The metrics you're interested in gathering for the application focus on user experiences, like connection errors and traffic. The stack for your telnet-server application will consist of three main monitoring applications and a traffic simulation application. You'll use these applications to monitor, alert, and visualize the metrics instrumented by telnet-server.

The monitoring applications are Prometheus, Alertmanager, and Grafana. They are commonly used in the Kubernetes ecosystem. *Prometheus* is a metric collection application that queries metric data with its powerful built-in query language. It can set alerts for those metrics as well. If a collected metric crosses a set threshold, Prometheus sends an alert to *Alertmanager*, which takes the alerts from Prometheus and decides where to route them based on some criteria that are user configurable. The routes are usually notifications. *Grafana* provides an easy-to-use interface to create and view dashboards and graphs from the data Prometheus provides. The traffic simulator, bbs-warrior, simulates the traffic an end user of the telnet-server application might generate. This lets you test your monitoring system, application metrics, and alerts. Figure 9-1 shows an overview of the example stack.

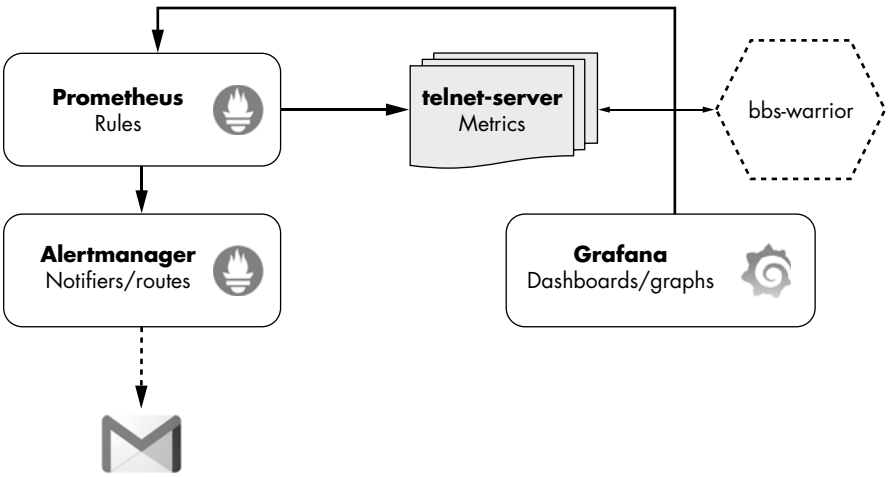


Figure 9-1: Overview of our monitoring stack

Installing the Monitoring Stack

To install these applications, you'll use the provided Kubernetes manifest files. The manifest files for the monitoring stack and traffic simulator are in the repository (*https://github.com/bradleyd/devops_for_the_desperate/*), under the *monitoring* directory. Within that directory are four subdirectories: *alertmanager*, *bbs-warrior*, *grafana*, and *prometheus*. These make up the example monitoring stack. You'll install Prometheus, Alertmanager, and Grafana in a new Kubernetes Namespace called `monitoring` by applying all the manifests in each of these directories.

In a terminal, enter the following command to install the monitoring stack and bbs-warrior:

```
$ minikube kubectl -- apply -R -f monitoring/
namespace/monitoring created
serviceaccount/alertmanager created
configmap/alertmanager-config created
deployment.apps/alertmanager created
service/alertmanager-service created
cronjob.batch/bbs-warrior created
configmap/grafana-dashboard-pods created
configmap/grafana-dashboard-telnet-server created
configmap/grafana-dashboards created
configmap/grafana-datasources created
deployment.apps/grafana created
service/grafana-service created
clusterrolebinding.rbac.authorization.k8s.io/kube-state-metrics created
clusterrole.rbac.authorization.k8s.io/kube-state-metrics created
deployment.apps/kube-state-metrics created
serviceaccount/kube-state-metrics created
service/kube-state-metrics created
clusterrole.rbac.authorization.k8s.io/prometheus created
clusterrolebinding.rbac.authorization.k8s.io/prometheus created
configmap/prometheus-server-conf created
deployment.apps/prometheus created
service/prometheus-service created
```

The output shows that all the manifests for your monitoring stack and bbs-warrior were run without errors. The `-R` flag makes the `kubectl` command recursively go through all the application directories and their subdirectories under the *monitoring* directory. Without this flag, `kubectl` will skip any nested subdirectories, like *grafana/dashboards/*. Prometheus, Grafana, Alertmanager, and bbs-warrior should be up and running in a few moments.

NOTE *You may have noticed that the* namespace *file in the* monitoring *directory has a* 00_ *prefix. This prefix ensures that when* `kubectl` *applies the manifests, the* namespace *file will be evaluated first. All the monitoring applications are installed in a separate Namespace called* `monitoring`*. Those applications reference the* `monitoring` *Namespace, and if it isn't created first, they will not install. Using a* 00_ *prefix is a simple way to force ordering in a directory of files. If you needed a file to be the second one evaluated, you would use a* 01_ *prefix.*

Verifying the Installation

If the monitoring stack installation was successful on your Kubernetes cluster, you should be able to access Grafana's, Alertmanager's, and Prometheus's web interfaces on your browser. In the provided Kubernetes manifest files, I have set the Kubernetes Service types for Prometheus, Grafana, and Alertmanager to `NodePort`. A Kubernetes *NodePort* Service allows you to connect to an application outside the Kubernetes cluster, so you should be able to access each application on the minikube IP address and a dynamic port. You should also be able to confirm that the bbs-warrior traffic simulator was installed and is running periodically.

Grafana

In a terminal, enter the following command to open Grafana:

```
$ minikube -n monitoring service grafana-service
|------------|-----------------|-------------|-----------------------------|
| NAMESPACE  |      NAME       | TARGET PORT |             URL             |
|------------|-----------------|-------------|-----------------------------|
| monitoring | grafana-service |        3000 | http://192.168.99.105:31517 |
|------------|-----------------|-------------|-----------------------------|
Opening service monitoring/grafana-service in default browser...
```

Grafana lives in the `monitoring` Namespace, so this command uses the `-n` (Namespace) flag to show the `minikube service` command where to locate the Service. If you omit the `-n` flag, minikube will error, as there's no Service named `grafana-service` in the default Namespace. You should now see Grafana open in your web browser, with the telnet-server dashboard loaded as the first page. If you don't see the telnet-server dashboard, check the terminal where you ran the `minikube service` command for any errors. (You'll need access to Grafana to follow along with the rest of this chapter.) We'll discuss the graphs on the Grafana dashboard later; for now, you should ensure that Grafana is installed correctly and that you can open it in your browser.

Alertmanager

In a terminal, enter the same command you used to open Grafana in your browser, but replace the Service name with **`alertmanager-service`**, like this:

```
$ minikube -n monitoring service alertmanager-service
--snip--
```

The Alertmanager application should now be open in your browser. This page has a few navigation links, like Alerts, Silences, Status, and Help. The Alerts page displays current alerts and any metadata, like timestamps and severity associated with an alert. The Silences page shows any alerts that have been silenced. You can mute or silence an alert for a specific amount of time, which is helpful if an alert is being triggered and you don't want

to keep getting paged for it. The Status page shows information about Alertmanager, like its version, ready status, and current configuration. Alertmanager is configured via the *configmap.yaml* file in the *alertmanager/* directory. (You'll edit this file later to enable notifications.) Finally, the Help page is a link to Alertmanager's documentation.

Prometheus

In your terminal, enter the same command you just entered, but replace `grafana-service` with **`prometheus-service`** to open Prometheus:

```
$ minikube -n monitoring service prometheus-service
--snip--
```

Prometheus should open in your browser with a few links at the top of the page: Alerts, Graph, Status, and Help. The Alerts page displays all known alerts and their current state. The Graph page is the default page that allows you to run queries against the metric database. The Status page contains information about Prometheus's health and configuration file. Prometheus, like Alertmanager, is controlled by the *configmap.yaml* file in the *prometheus* directory. This file controls what endpoints Prometheus scrapes for metrics, and it contains the alert rules for specific metrics. (We'll explore the alert rules later.) The Help page is a link to Prometheus's official documentation. For now, you are just confirming that Prometheus is running. Leave Prometheus open, as you'll need it in the next section.

bbs-warrior

The bbs-warrior application is a Kubernetes CronJob that runs every minute and creates a random number of connections and errors to the telnet-server application. It also sends a random number of BBS commands (like `date` and `help`) to the telnet-server, to mimic typical user activity. About a minute after you install bbs-warrior, it should start generating random traffic. This simulation should last only a few seconds.

To make sure bbs-warrior is active and installed correctly in your Kubernetes cluster, enter the following command in a terminal:

```
$ minikube kubectl -- get cronjobs.batch -l app=bbs-warrior
NAME                SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
bbs-warrior         */1 * * * *   False     0        25s             60s
```

The `-l` (label) flag narrows down the results when searching for CronJobs. The output shows that the CronJob was installed over a minute ago (`60s`, under the `AGE` column) and that it last ran `25` seconds ago (under the `LAST SCHEDULE` column). If it were actively running, the `ACTIVE` column would be set to `1` rather than `0`.

You now know that the CronJob ran, but you should make sure it completed successfully. To do that, you'll list the Pod with the label `bbs-warrior`

in the default Namespace and look for `Completed` in the `STATUS` column. In the same terminal you used above, enter the following command:

```
$ minikube kubectl -- get pods -l app=bbs-warrior
NAME                            READY   STATUS              RESTARTS   AGE
bbs-warrior-1600646880-chkbw    0/1     Completed           0          60s
```

The output shows that the `bbs-warrior` CronJob completed successfully about `60` seconds ago. If the CronJob has a status different from `Completed`, check the Pod's logs for errors like you did in Chapter 7.

Metrics

You've installed and verified your monitoring stack, so now, you should focus on what you are monitoring for your telnet-server. Since you want to tailor your metrics to user happiness, you should use a common pattern to align all your applications. This is always a good approach when instrumenting your services, because allowing applications to do their own unique version of metrics makes triaging (and thus, on-call shifts) very difficult.

For this example, you'll explore a common metric pattern called *Golden Signals*. This provides a subset of metrics to track, like errors and traffic, plus a common language for you and your peers to use to discuss what healthy looks like.

Golden Signals

The Golden Signals (a term first coined by Google) are four metrics that help us understand the health of a microservice. The Golden Signals are latency, traffic, errors, and saturation. *Latency* is the time it takes for a service to process a request. *Traffic* is how many requests an application is receiving. *Errors* refers to the number of errors an application is reporting (such as a web server reporting 500s). *Saturation* is how full a service is. For a saturation signal, you could measure CPU usage to determine how much headroom is left on the system before the application or host becomes slow or unresponsive. You will use this pattern often when measuring applications. If you are ever in a situation where you don't know what to monitor, start with the Golden Signals. They'll provide ample information about your application's health.

A *microservice* typically is an application loosely coupled to other services in your platform. It is designed to focus only on one or two aspects of your overall domain. In this chapter, the telnet-server application will serve as the microservice whose health you will measure.

Adjusting the Monitoring Pattern

Chances are your application will not fit neatly into a predefined monitoring pattern like the Golden Signals. Use your best judgment about what matters. For example, I decided not to track latency when instrumenting

the telnet-server application even though the pattern lists it. Users of such an application typically wouldn't connect, run a command, and then quit. You could track latency of the commands, or you could add tracing for each command workflow. However, that would be overkill for this sample application and beyond the scope of this book. Your commands are for demonstration purposes only, so focusing on traffic, errors, and saturation signals will provide an overall idea of the application's health from a user's point of view.

OTHER METRIC PATTERNS

Two other common metric patterns are RED and USE. The *RED* (rate, error, and duration) method *(https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture/)* was created by Tom Wilkie of Grafana Labs. Like Golden Signals, RED was designed to help monitor microservices. However, RED focuses more on the application's health than on underlying system resources like CPU or memory. The *rate* is the number of requests per second a service is receiving. *Error* is the number of failed requests per second (such as connection failures that a client experiences) that the service encounters. *Duration* is the amount of time it takes to serve a request, or how long it takes to return the data requested from your service to the client.

The *USE* (utilization, saturation, and errors) method was developed by Brendan Gregg (*https://www.brendangregg.com/usemethod.html*) for quickly discovering performance issues based on underlying resources rather than the microservices that run on them. *Utilization* is the average time the resource (for example, a disk drive at 85 percent usage) is busy doing work. *Saturation* can be thought of as extra work the system could not get to, such as happens with a busy host that is queueing up connections to serve traffic. *Errors* are the number of errors (such as network collisions or disk IO errors) a system is having.

The telnet-server Dashboard

Let's review the traffic, saturation, and error signals on your Grafana dashboard. In the browser where you first opened Grafana, the telnet-server dashboard has three graphs for the Golden Signals and two collapsed graph rows titled System and Application (see Figure 9-2). You'll focus on the Golden Signals graphs, which are as follows: Connections per second, Saturation, and Errors per second.

The first graph, Connections per second (in the top left), provides the traffic Golden Signal. In this case, you measure how many connections per second you are receiving in a two-minute time frame. The telnet-server application increases a metric counter each time a connection is made, providing a good idea of how many people connect to the application. Many connections could pose an issue with performance or reliability. In this example, the x-axis shoots up over 4.0 connections per second for both

telnet-server Pods. Your graphs will show different results from mine since bbs-warrior generates the traffic randomly; the goal is to make sure the graphs are being populated.

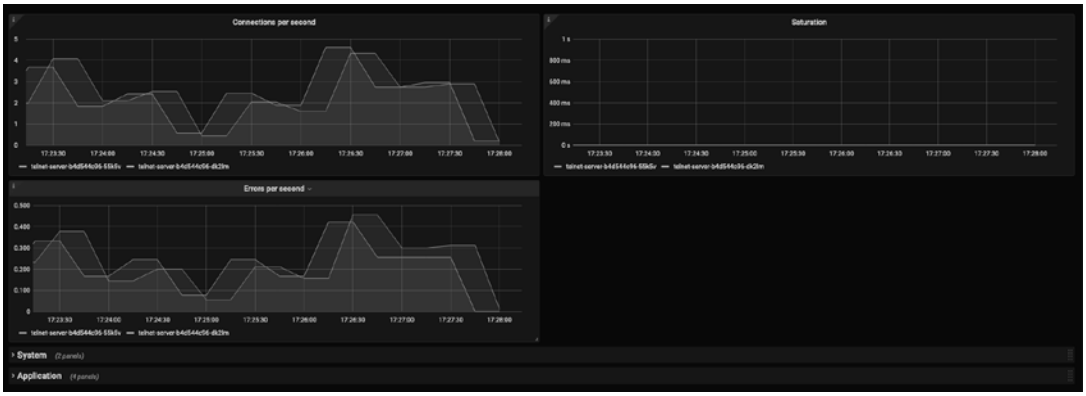


Figure 9-2: The telnet-server Grafana dashboard

The Saturation graph (top right) represents the saturation Golden Signal. For saturation, you measure the amount of time Kubernetes throttles your telnet-server container's CPU. You set a CPU resource limit of 500 millicpu for the telnet-server container in Chapter 7. Therefore, if the telnet-server container uses more than the maximum limit, Kubernetes will throttle it, possibly making the telnet-server slow to respond to commands or connections. This throttle could potentially cause poor performance or a service interruption. In the Saturation graph shown in Figure 9-2, the x-axis is flat at 0 microseconds for both Pods. The line of the graph will rise if CPU throttling occurs.

The Errors per second (bottom left) graph maps to the error Golden Signal. For this metric, you track the connection errors per second that you receive in a two-minute time frame. These errors are incremented when a client fails to connect properly or if the connection is killed unexpectedly. A high error rate could indicate a code or infrastructure issue that you need to address. In the graph shown in Figure 9-2, the errors-per-second rate is spiking to 0.4 for both pods.

The collapsed two rows at the bottom of this dashboard contain some miscellaneous graphs not covered in this chapter, but you should explore them on your own. The telnet-server dashboard System row contains two graphs: one for memory, and one for CPU usage by the telnet-server Pods. The Application row contains four graphs: Total Connections, Active Connections, Connection Error Total, and Unknown Command Total.

The telnet-server dashboard is in the *grafana/dashboards/telnet-server.yaml* file. This file is a Kubernetes ConfigMap resource that contains the JSON configuration that Grafana requires to create the dashboard and graphs.

NOTE *Keep your dashboards under version control to make it easier to reproduce and make changes.*

PromQL: A Primer

PromQL is a query language built into the Prometheus application. You use it to query and manipulate metric data. Think of PromQL as a distant cousin of SQL. It has some built-in functions (like `average` and `sum`) to make querying data easier plus conditional logic (like `>` or `=`). We won't explore this query language in depth here except to show how I queried telnet-server's Golden Signal metrics to populate your graphs and alerts.

For example, here is the query you enter to generate the Errors-per-second graph:

```
rate(telnet_server_connection_errors_total{job="kubernetes-pods"}[2m])
```

The name of the metric is `telnet_server_connection_errors_total`. This metric measures the total amount of `connection errors` a user may encounter. The query uses Prometheus' `rate()` function, which calculates the per-second connection error average over a specified time interval. You limit the time frame for which this query fetches data to two minutes using square brackets `[2m]`. The result will show the two running telnet-server Pods you installed in Chapter 7. The curly brackets ({}) allow you to refine the query, using labels as matchers. Here, you specify that you want data only for the `telnet_server_connection_errors` metric with the `job="kubernetes-pods"` label.

When creating an alert rule in Prometheus, you can enter the same query as above to drive the alert. However, this time, you should wrap the results from the `rate()` function in a `sum()` function. You'll do this because you want to know the overall error rate for both Pods. The alert rule should look like the following:

```
sum(rate(telnet_server_connection_errors_total{job="kubernetes-pods"}[2m])) > 2
```

At the end of the query, you add greater-than (`>`) conditional logic with a number: `2`. This basically means that if the error rate is greater than two per second, this alert query evaluates to true. (Later in this chapter, we'll discuss what happens when alert rules are true.)

If you want to review or tinker with any of these metrics, see the Graph page in the Prometheus web interface. Figure 9-3 shows the `telnet_server_connection_errors_total` query being run.

Figure 9-3: Running a query in Prometheus's web interface

The query returns connection error data for both Pods. To learn more about PromQL, visit *https://prometheus.io/docs/prometheus/latest/querying/basics/* for more examples and information.

NOTE *A good rule of thumb when calculating per-second averages (rates) is to set the sample time window to at least two times the Prometheus scrape interval. In your case, Prometheus fetches the telnet-server metrics endpoint every 30 seconds, so your time interval should not be less than one minute.*

Alerts

Metrics and graphs only constitute half of a monitoring solution. When your application decides to take a stroll off a cliff (and it will), someone or something needs to know about it. If a Pod dies in a Deployment, Kubernetes just replaces it with a new one. But if the Pod keeps restarting, someone needs to address it, and that's where the alerts and notifications come into play.

What constitutes a good alert? Besides an alert for each of your application's Golden Signals, you may need an alert around a key metric to monitor. When this does happen, keep in mind a couple of guidelines to follow when creating alerts:

Do not set thresholds too low. Setting alert thresholds too low can cause the alerts to repeatedly fire and then clear if you have spiky metrics. This behavior is known as *flapping*, and it can be quite normal. The system should not issue alerts for flapping metrics every few minutes, because on-call engineers get stressed out when they repeatedly get a notification and then find the alarm has already cleared.

Avoid creating alerts that are not actionable. Don't create alerts for a service when nothing can be done to remedy it. I call these alerts *status alerts*. Nothing is more frustrating to an on-call engineer than being woken up in the middle of the night only to babysit an alert that requires no action.

For this book, I have provided three alerts called `HighErrorRatePerSecond`, `HighConnectionRatePerSecond`, and `HighCPUThrottleRate` (more on these later). These alerts are located in the *prometheus.rules* section inside Prometheus's configuration file (*configmap.yaml*). Prometheus uses alert rules to decide whether a metric is in an undesired state. An *alert rule* has information like the alert name, PromQL query, threshold, and labels. For your example, I have gone against my own alert-creation advice and set the provided thresholds extremely low, allowing bbs-warrior to trigger the alerts easily. Nothing beats a live example when learning about real-time metrics and alerts!

Reviewing Golden Signal Alerts in Prometheus

You can view alerts in either Prometheus's or Alertmanager's web interfaces. The difference is that Alertmanger displays only alerts that are being triggered, whereas Prometheus will show all alerts, whether they are firing or not. You want to view all the alerts, so you'll use Prometheus for this example. However, you should visit Alertmanager's interface as well when an alert is being triggered.

In the browser where Prometheus was originally opened, click the **Alerts** link in the top-left navigation bar. You should see the three provided telnet-server Golden Signals alerts: `HighErrorRatePerSecond`, `HighConnectionRatePerSecond`, and `HighCPUThrottleRate`. These alerts were created when you installed Prometheus earlier. The Alerts page should look like Figure 9-4.

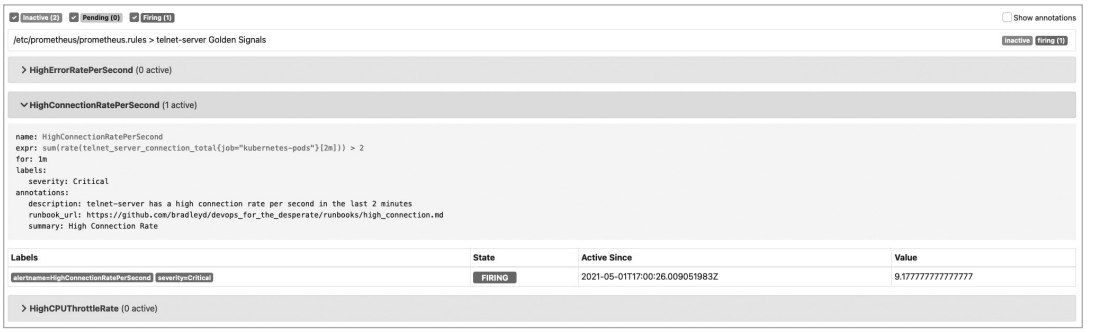


Figure 9-4: Prometheus alerts for telnet-server

Each alert will be in one of three states: Pending (yellow), Inactive (green), or Firing (red). In Figure 9-4, the `HighConnectionRatePerSecond` alert is Firing. The other two alerts, `HighCPUThrottleRate` and `HighErrorRatePerSecond`, are Inactive since they are not being triggered. Your Alerts page will be different from mine because of bbs-warrior's randomness. If your page doesn't show any alerts in a Firing state, wait a few minutes until more traffic is generated. Then refresh the browser page. In all my testing for this chapter, I always had at least one alert transition to a Firing state.

The `HighErrorRatePerSecond` alert is concerned with the number of connection errors received per second. If the rate of connection errors in a two-minute window is greater than 2, the alert is in a Firing state. On my local Kubernetes setup, the alert is currently in the Inactive state.

The next alert, `HighConnectionRatePerSecond`, detects whether the connection rate is greater than 2 per second in a two-minute time frame. Currently, this alert is in the Firing state. Figure 9-4 shows that the current value for my connection rate is more than 9.1 connections per second, which is well beyond the set threshold of 2. I have expanded the alert in the browser to show the provided metadata in a key-value layout that an alert provides. In the labels section for all three alerts, I have set a label called `severity` with a value of `Critical` so it's easier to distinguish between noncritical alerts and ones that need immediate attention. You'll use this label to route important alerts in Alertmanager later. The annotations section includes a description, a summary, and a link to a *runbook*, which is a blueprint that provides unfamiliar engineers with the what, why, and how for a service. Having this information is crucial when sending out an alert notification, because it gives the person on call an idea of what to look for when troubleshooting.

The last alert, `HighCPUThrottleRate`, detects high CPU saturation. If the CPU is being throttled by Kubernetes for more than 300 microseconds in a two-minute window, you'll transition to a Firing state. This alert is currently inactive, but normally, I'd suggest a minimum five-minute window when tracking CPU throttling. This is because smaller time windows can make you more susceptible to alerting on a temporarily spiky workload.

Routing and Notifications

You've verified that the metrics and alerts are visible and active, so now, you should set up Alertmanager to send out email notifications. Once an alert is in the Firing state, Prometheus sends it to Alertmanager for routing and notification. Notifications can be sent via text messages, push notifications, or email. Alertmanager calls these notification methods *receivers*. Routing is used to match on alerts and send them to a specific receiver. A common pattern is to route alerts based on specific labels. Alert labels are set in the Prometheus *configmap.yaml* file. You'll use this pattern later, when you enable notifications.

The provided Alertmanager configuration is located in the *alertmanager/configmap.yaml* file. It is set up to match on all alerts with a `severity` label set to `Critical` and route them to a *none receiver*, which is basically a black hole that won't notify anyone when there's an alert. This means that to see whether an alert is being triggered, you would need to visit the web page on either Alertmanager or Prometheus. This setup isn't ideal, as refreshing the web browser every few minutes would become tedious, so you'll route any alert to the `email` receiver if the alert has a `severity` label set to `Critical`. If you're following along, this step is completely optional, but it shows you how to configure receivers in Alertmanager.

Enabling Email Notifications

To route an alert to the `email` receiver, you need to edit Alertmanager's configuration. I have stubbed out a template for the `email` receiver and `route` block in the *configmap.yaml* file. The email example is based on a Gmail account, but you can alter it to accommodate any email provider. See *https://www.prometheus.io/docs/alerting/latest/configuration/#email_config/* for more details.

Open Alertmanager's *configmap.yaml* file in your favorite editor; it should look like this:

```
--snip--
    global: null
    receivers:
  ❶ #- name: email
    #  email_configs:
    #  - send_resolved: true
    #    to: <GMAIL_USERNAME@gmail.com>
    #    from: <GMAIL_USERNAME@gmail.com>
    #    smarthost: smtp.gmail.com:587
    #    auth_username: <GMAIL_USERNAME@gmail.com>
```

```
    #    auth_identity: <GMAIL_USERNAME@gmail.com>
    #    auth_password: <GMAIL_PASSWORD>
  ❷ - name: none
    route:
      group_by:
        - job
      group_interval: 5m
      group_wait: 10s
    ❸ receiver: none
      repeat_interval: 3h
      routes:
      ❹ - receiver: none
          match:
            severity: "Critical"
```

Here, you have two receivers named `email` ❶ and `none` ❷. The `none` receiver won't send alerts anywhere, but when uncommented, the `email` receiver will send alerts to a Gmail account. Uncomment the `email` receiver lines and then replace with an email account you can use for testing.

NOTE *If you are using Gmail and have 2FA enabled, you'll need to set up an app-specific password credential as the generic username. Password authentication will not work. See* https://support.google.com/accounts/answer/185833/ *for more details.*

After configuring your email settings, change the `receiver` ❸ under the `routes` section to **`email`**. This configures Alertmanager to route any alert to the `email` receiver if the alert has a `severity` label set to `Critical`. The receiver line ❹ should now look like this:

```
- receiver: email
```

You'll still have your default or catch-all receiver ❸ set to `none`, so any alert that does not match your `severity` label rule will be sent there. Save this file, as you are done modifying it.

Applying Alertmanager's Configuration Changes

Next, you'll update Alertmanager's ConfigMap inside the Kubernetes cluster. Since the local file contains changes that don't exist on the cluster, enter the following in a terminal:

```
$ minikube kubectl -- apply -f monitoring/alertmanager/configmap.yaml
configmap/alertmanager-config configured
```

The next step is to tell Kubernetes to restart the Alertmanager Deployment so it can pick up the new configuration changes. In the same terminal, enter the following command to restart Alertmanager:

```
$ minikube kubectl -- -n monitoring rollout restart deployment alertmanager
deployment.apps/alertmanager restarted
```

The Alertmanager Pod should restart after a few moments. If you have any alerts in the Firing state, you should start receiving email in your inbox. Depending on Alertmanager and your email provider, the notifications may take some time to appear.

If you do not receive any notification emails, check for a couple of common issues. First, make sure the *configmap.yaml* file does not have any typos or indentation errors. It is very easy to misalign a YAML file. Second, make sure the email settings you entered match what is required by your email provider. Look in Alertmanager's logs to find these and other common issues. Enter the following `kubectl` command to view the logs for any errors:

```
$ minikube kubectl -- -n monitoring logs -l app=alertmanager
```

If you need to disable the notifications for any reason, set the routes receiver back to `none` in the *configmap.yaml* file, apply the manifest changes, and restart.

You now have alerts and notifications configured for telnet-server's Golden Signals.

Summary

Metrics and alerts are foundational pieces when monitoring an application. They provide insight into the health and performance of your service. In this chapter, you learned about the Golden Signals monitoring pattern and how to install a modern monitoring stack inside a Kubernetes cluster using Prometheus, Alertmanager, and Grafana. Finally, you learned how to configure Alertmanager to send email notifications for critical alerts.

In the next chapter, we'll shift gears and discuss common troubleshooting scenarios you will find on a host or network, plus the tools you can use to diagnose them.

10

TROUBLESHOOTING HOSTS

Engineers spend a lot of time trying to figure out why something isn't working as intended. Instrumentation, tracing, and monitoring play big roles in determining the health of a host or application, but sometimes, observability is not enough. There will be times when you'll need to roll up your sleeves and figure out why something is broken and how to fix it. In other words, you'll be troubleshooting and debugging. *Troubleshooting* is the process of analyzing the system and rooting out potential causes of trouble. *Debugging*, on the other hand, is the process of discovering the cause of trouble and possibly implementing steps to remedy it. The differences are subtle, and in fact, you can think of debugging as a subset of troubleshooting. Most of what you'll do in this chapter is considered troubleshooting.

In this chapter, you'll explore common performance problems and issues you may encounter on a Linux host. You'll look at symptoms, commands you can use to diagnose various potential problems, and the next steps to take after troubleshooting. By the end of this chapter, you'll have expanded your command line arsenal and sleuthing skills to troubleshoot common issues.

NOTE *All scenarios in this chapter are geared toward Linux. If you are using another OS, like macOS, these concepts might cross over, but the tools can behave differently. Check the tools' documentation for your OS for any potential differences. I mostly used tools that were installed by default, but you'll need to install some of them using your local package manager. The tools I chose have some overlap in some cases, but I wanted to give you a variety to make you comfortable with different ways to poke a host.*

Troubleshooting and Debugging: A Primer

Troubleshooting and debugging is an art, not an exact science. Rarely will you see a big neon sign with an arrow pointing to the exact issue. Most of the time, you'll find a trail of breadcrumbs that leads you from clue to clue. You may have to crawl through the weeds to find those crumbs, and you may want to pull out your hair before you find what you're looking for. But diagnosing a broken system can be very rewarding, and figuring out an issue that's plaguing your customers or haunting a coworker can feel amazing.

But even an artist needs a method, and having a standard set of steps and techniques to follow whenever you are investigating an issue is a great way to start. So here are some tips to keep in mind when venturing forth to confront those fickle beasts we call hosts:

Start simple. When troubleshooting a problem, it can be tempting to jump to conclusions and assume it's the worst-case scenario. Instead, be methodical and build upon the knowledge you have gained. The problem is usually human error.

Build a mental model. Understanding what the system's role is and how it interacts with other systems will help you troubleshoot faster. You will find yourself spending less time worrying about architecture and more time working on the issue.

Take your time developing a theory. You may want to latch on to the first clue you find, but it's always worth checking to see if the breadcrumb trail leads any farther. Come up with a test to validate your theory.

Have consistent tools across hosts. Make sure your hosts were built with the same tooling. There is nothing worse than logging in to a host and finding out it is not like the others. Tool consistency is one of the benefits of building your hosts with automation.

Keep a journal. Keep a high-level account of problems, symptoms, and fixes so you don't forget important details about an issue. Your future self will thank you.

Know when to ask for help. If your business depends on solving an issue but you are struggling to find the cause, it is best to send up a flare. Someone with more experience can usually help, and someday, you will pay that knowledge forward or maybe even return the favor.

NOTE *All the commands used in this chapter have many more use cases and a plethora of parameters and flags they can accept. If you are unsure about a flag or want to learn more, visit the command's man pages for more information.*

Scenario: High Load Average

Linux has a metric called *load average* that provides an idea of how busy a host is. The load average takes into account data like CPU and I/O when calculating this number. The load of a system is displayed in 1-minute, 5-minute, and 15-minute averages. At first glance, any high number in an average might seem like a problem. But troubleshooting a high load average can be tricky because a high load doesn't always indicate that your host is in a degraded state. A busy host can have a high load but still respond to requests and commands without issue. It's like when two people have the same temperature, but one person is awake and functioning in a normal capacity and the other is bedridden and lethargic. Each host and workload is different, so you first need to identify what a normal range for your host looks like. A good rule of thumb is if the load average is larger than the CPU core count, you may have processes waiting and causing latency or performance degradation. When investigating this scenario, a good first step is to identify the high load and try to locate any process that could be causing it.

uptime

Enter the **`uptime`** command to display how long a host has been running, the number of logged-in users, and the system load. It reports the load in 1-minute, 5-minute, and 15-minute averages:

```
$ uptime
09:30:38 up 47 days, 31 min, 2 users, load average: 8.05, 1.01, 0.00
```

This four-core CPU host has been `up` for `47 days` and `31 minutes`, and `2 users` are currently logged in. The 1-minute `load average` is `8.05`. The 5-minute `load average` is `1.01`, which means the pressure on the system has been increasing during somewhere between 1 and 5 minutes of runtime. You know this because the 15-minute `load average` is `0.00` (no load at that time). If the numbers were reversed, with the 15-minute load showing

the higher number and the 1-minute load at zero, you could infer that the spike in load is not ongoing and happened around 15 minutes ago. Since this load seems to be increasing and has been climbing for more than 5 minutes, and since it is greater than the CPU core count, it may be worth investigating why.

top

The `top` command displays information about a system and the processes running on that host. It provides details like CPU percentage, load average, memory, and process information. Execute the **`top`** command to launch an interactive real-time dashboard showing system information, as shown in Figure 10-1.

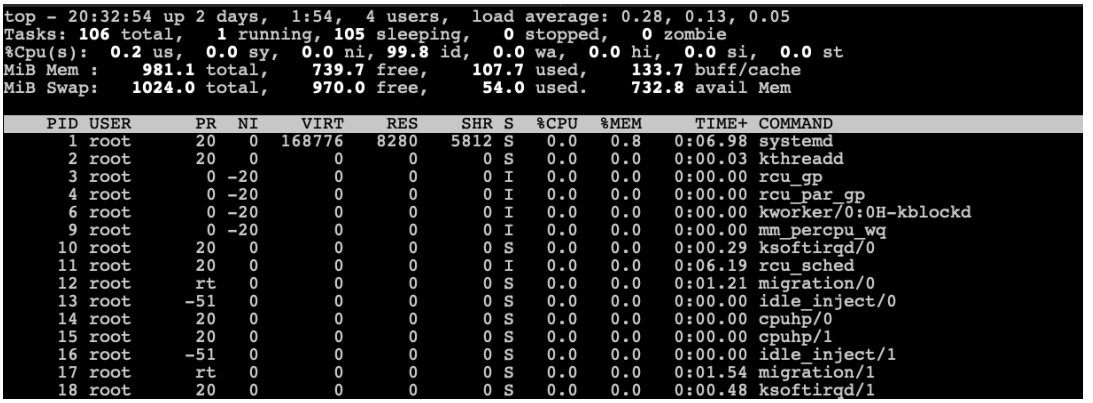


Figure 10-1: The top command output on a mostly idle host

By default, `top` sorts all the processes by `CPU` percentage. The first row contains the process using the most `CPU` percentage at that given poll cycle. The display refreshes (polls) every 3.0 seconds, so you'll want to view `top` for a few cycles before settling on a process or any data that might be or indicate the cause of the high load.

The following snippet is from a `top` report where a process is using 120 percent CPU:

```
PID  USER   ...    RES   SHR  S  %CPU  %MEM     TIME+  COMMAND
3048 root   ... 177740  5164  S 120.3   1.8 173:02.78  fail2ban-server
```

The key columns are `PID`, `RES`, `%CPU`, `%MEM`, and `COMMAND`. (Others are omitted here for readability.) The `fail2ban-server` command (in the `COMMAND` column) is using 120.3 percent CPU and is consuming around `177,740`KB of memory, as shown in the `RES` column. This process is using around `1.8` percent of the total memory (`%MEM`) available on the host. Taking everything into account, it would be a good idea for you to investigate process `3048` to determine why it is using so much CPU.

Next Steps

In a scenario with a high load average, you'll want to dig down further into the offending process. Perhaps this application is misconfigured, hung, or busy waiting on external resources (like a disk or an HTTP call). Maybe the host is undersized for its use case. If it's a cloud-based instance, perhaps there aren't enough CPU cores or disk IOPS. Also, check whether the host is experiencing increased traffic during this time, as that could indicate an intermittent spike. You can also use tools like `vmstat`, `strace`, and `lsof` to discover more about a process's interaction with the system. (You'll learn more details about those tools in later sections.)

Scenario: High Memory Usage

Temporary spikes in traffic, performance-related issues, or an application with a memory leak can cause memory to be consumed at a high rate. The first step in investigating high memory usage is to make sure the host is really running low on memory. Linux likes to use all the memory for caches and buffers, so it can appear that free memory is low. But the Linux kernel can reallocate that cached memory elsewhere if needed. The `free`, `vmstat`, and `ps` commands can help identify how much memory is being used and what process may be the culprit.

free

The `free` command provides a quick sanity check on system memory by displaying used and available memory at the time it is run. Pass the `-h` and `-m` flags to instruct the `free` command to show all output fields in human-readable (`-h`) format using the *mebibyte* unit (`-m`) of measure. In *human-readable format*, data appears in familiar units like *mebibyte* or *gibibyte* instead of bytes. The following example shows a host that's low on available memory. Enter the following command to display memory:

```
$ free -hm
              total        used        free      shared  buff/cache available
Mem:          981Mi       838Mi        95Mi       3.0Mi        47Mi       43Mi
Swap:         1.0Gi       141Mi       882Mi
```

The system contains `981Mi` of `total` memory, and `838Mi` of memory is being `used`, with `95Mi` `free`. The `buff/cache` column contains information from data that has been read off disk and the metadata associated with it. This is used for fast retrieval if you need to access it again, which is why Linux tries to use all the system memory it can instead of letting it sit idle. A Linux host will swap data out of memory and write it to disk if a system is running low on memory. As you can imagine, using disk as memory is much slower than using actual RAM. If the `free` column for `Swap` is ever low, your system may be performing slower than it normally can. In this example, the system is swapping to disk only a little (`141Mi`), which can be normal.

The used and free columns can be misleading on a Linux host. Linux likes to use every bit of RAM on a system, so it may appear at a quick glance that a host is low on memory. Or, as in this case, it can appear that there is more memory than actually is available. Here, the free column shows 95Mi, but according to the available column, only 43Mi is left. When using the free command to display system memory, pay attention to the available column as a barometer of actual memory available to the system and new processes.

Looking at how little memory is available in this example, it's safe to say this host has a memory shortage. Having roughly 43Mi out of 1Gi left on a system can cause stability issues and stop new processes from being created. It can also force the Linux kernel to invoke the out of memory manager (OOM) and select a process to kill, which can and will cause unexpected behavior.

vmstat

The vmstat command provides useful information about processes, memory, IO, disks, and CPU activity. It can report this data over a period of time, which is an upgrade over the free command and makes trends much easier to spot. You'll pass two parameters to the vmstat command: delay, which specifies the time delay between each of the polling counts, and count, which specifies the number of times vmstat will fetch data until it quits. For this example, you will poll the data five times with a one-second delay between each poll. Enter the following command to poll the data:

```
$ vmstat 1 5
procs ---------memory--------  --swap-- -----io---- -system- ---cpu--------
 r b   swpd   free buff  cache  si   so   bi    bo   in   cs us sy id wa st
 2 0  54392  74068 7260 117804   0   10   84   432   81  158  3  1 96  0  0
 1 0  54392  73864 7260 117852   0    0    8     0  379  104 44  0 56  0  0
 1 2  54392  71768  484  38724 104    0  496   196  469  327 41  1 57  1  0
 1 0  54392  71508  484  39768  20    0 1024     0  357   82 44  0 56  0  0
 1 0  54392  71508  484  39768   4    0    0     0  370   43 46  0 54  0  0
```

The vmstat report is divided into multiple categories: procs, memory, swap, io, system, and cpu. Each category contains like columns. The first row of data is an average of each statistic since the last boot time. Since you are hunting for high memory usage, you'll focus only on the memory and swap sections from the vmstat output.

The swpd column of the memory section shows the total swap space used; in this case, it's around 54Mi (54,392Ki). Next comes the free column. According to vmstat, the free memory has fluctuated between 71,000Ki and 74,000Ki in the polling snapshot. This does not mean you have only 71,000Ki of memory available; it's an estimate because of the free-able cache and buffers.

Under the swap section are two columns: si (swapped in) and so (swapped out). The si and so columns indicate you are paging memory to and from the disk. At one point, you were swapping memory from the disk at about 104KiB per second. As mentioned previously, a little swapping can be okay, but being low on free memory plus swapping usually indicates a memory bottleneck.

The r and b columns under procs can provide good indications of possible bottlenecks. The r column is the number of running (or waiting-to-run) processes. A high number here can indicate a CPU bottleneck. The b column is the number of processes in an uninterruptable sleep. If the number in the b column is high, it can be a good signal that there are processes waiting on resources like disk or network IO.

ps

If memory usage is high on the host, you'll want to check all the running processes to find where the memory is being used. The ps command provides a snapshot of the current processes on a host. You'll use some flags to narrow down the results and show only the top-10 hosts sorted by most memory. Enter the following command:

```
$ ps -efly --sort=-rss | head
S UID   PID PPID  C PRI  NI    RSS    SZ WCHAN STIME TTY TIME CMD
R root  931 930  93  80   0 890652 209077 -   05:56 ?   ...   memory-hog
S root  469   1   0 -40   - 18212 86454 -     Jan16 ?   ...  /sbin/multipathd
S root  672   1   0  80   0 10420 233460 -    Jan16 ?   ...  /usr/lib/snapd
S root  350   1   0  79  -1  7416 12919 -     Jan16 ?   ...  /lib/systemd
```

The -efly and --sort=-rss flags are used to show all the processes in a long format. The RSS (resident set size) column shows the amount of non-swappable physical memory a process uses (in kilobytes), in descending numerical order. You pipe those results to the head command, which displays only 10 by default. The CMD column shows the command that belongs to each process. In this example, the memory-hog command is using around 890MB (890,652KB) of physical memory, according to the RSS column. Considering that this host has only 1Gi of total memory, that application is hogging all the memory.

Next Steps

The steps you'll take to resolve a high-memory-usage issue like this will depend on risk factors for your system and/or users. If you're dealing with a production system, you'll want to tread lightly and check the logs, traces, and metrics to determine when and where the problem started. If this were a new behavior on a production system, rolling back memory-hog to a previous version would be a great first step. (Any time you can recover quickly in production is a win.) Once you have remediated the issue in production, do a performance profile in a different environment and dig through the clues to figure out why and where the memory is being used.

Scenario: High iowait

A host that is spending too much time waiting for disk I/O is said to have a condition called *high iowait*. The way to measure iowait is to check the percentage of time that CPUs are idle because the system has unfinished disk

I/O requests that are blocking processes from doing other work. Significant iowait usually results in a host having an increased load and possibly higher reported CPU usage than it normally would. To put it another way, if your CPU is waiting for the disk to respond, it has less time to service other requests from other parts of the system. One cause of high iowait might be an aging, slow, or failing disk. Another culprit could be an application that is performing heavy disk reads and writes. If you are in a virtualized environment, slow network-attached storage is most likely where your congestion lies.

All systems will have some iowait, and modern CPUs are faster than storage. High iowait by itself, however, is not enough to signal a problem. Some systems with high iowait can perform without issues, while others will show significant signs of a bottleneck. The goal is to identify issues that are accompanied by high iowait. There's no bright line with normal iowait on one side and high iowait on the other, so I have set the threshold for high iowait at anything over 30 percent that is sustained over a significant period.

Two command line tools, `iostat` and `iotop`, will help you troubleshoot a host with high iowait.

iostat

The `iostat` command line tool reports CPU and I/O stats for devices, so it's a great tool to help you determine whether your system is experiencing any iowait. If `iostat` is not installed by default, use your package manager to install the sysstat package.

As I mentioned previously, having some iowait is normal. You are looking for abnormal behavior, so you'll want to poll the system over a period of time to get a better view of the problem, like you did with the `vmstat` command. For this example, enter the command below to poll for statistics every second for a total of 20 times. The command and output should look like the following:

```
$ iostat -xz 1 20
--snip--
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           6.25    0.00   27.08   66.67    0.00    0.00

Device             r/s    rkB/s      w/s      wkB/s    %util ...
vda               0.00     0.00  1179.00  712388.00   100.00 ...
```

The first report `iostat` prints is from the last time the host was booted. Since that data is not relevant to your current troubleshooting scenario, I've omitted it here, along with multiple columns from the `Device` output. The `-xz` flag shows only active devices using an extended stat format. The `w/s` column shows that the `vda` device is executing a lot of write requests per second (`1179.00`). The `CPU` is waiting on outstanding disk requests around `66.67%` of the time (`%iowait`). Finally, as further proof that this disk is quite busy, the `%util` (percent utilization) column shows `100%`.

You can conclude that the host is suffering from high iowait that is sustained and not just intermittent. More importantly, you know that the

iowait is occurring on the device named vda. From here, it is worth trying to find a process that could be the cause of the increased iowait. You can do that with the iotop command, which you'll explore next.

iotop

The iotop command displays I/O usage in a top-like format. Not only does it provide an overview of I/O on the host, but it lets you drill down to the process level to locate any processes that might be causing a lot of disk I/O. Most distributions don't include iotop by default, so use your package manager to install it.

When running iotop, you'll want to limit the output to show only active processes that are performing I/O, using a batch mode that polls constantly to keep the output concise and reveal any possible I/O patterns. This command requires elevated permissions, so you'll need to run it with sudo or as a privileged user. Enter the command below:

```
$ sudo iotop -oPab
Total DISK READ:        15.04 M/s | Total DISK WRITE:     446.28 M/s
Current DISK READ:      15.04 M/s | Current DISK WRITE:   321.58 M/s
    PID  PRIO  USER   DISK READ  DISK WRITE  SWAPIN     IO  COMMAND
  88576 be/4 bob      512.00 M    616.81 M  0.00 % 83.26%  heavy-io
    469 rt/4 root       0.00 B      0.00 B  0.00 %  0.00%  multipathd -d -s
  --snip--
```

The -oPab flags make iotop show only processes performing I/O with accumulative stats in a batch mode. In this example, the heavy-io command is at 83.26%, according to the IO column. The PID column reports the process ID, which in this case is 88576. No other processes in your report are using a lot of I/O, so it's safe to assume that the heavy-io process is part of the reason for the high iowait.

Next Steps

After checking the stats and finding the process ID that is causing high iowait, you might want to explore what this application is used for. If you have the source code or configuration files, look for more clues by checking any disk operations or files the process has access to. Another cause for high iowait could be that your VM is in a cloud provider and you do not have enough provisioned I/O operations for your disk. Check the disk metrics to confirm and adjust the number to compensate the load. If all else fails, use tools like lsof to examine what files are open, strace to trace any system calls the process is making, or dmesg for any hardware kernel errors. (We'll discuss lsof, strace, and dmesg later in this chapter.)

Scenario: Hostname Resolution Failure

Traditionally, when a service needs to connect to another service, it uses Domain Name System (DNS) to look up the IP address to send it a request.

DNS is a directory for host IP address mappings. It allows us to use names like google.com or nostarch.com without needing to know those hosts' exact IP addresses. Humans are far better at remembering names than IP addresses like 142.250.72.78 or 104.20.208.3. Imagine if you had to find a store by trying to remember its latitude and longitude coordinates without using GPS instead of just remembering it's at 123 Main Street. You would get lost . . . a lot.

For this scenario, say you have an application that is trying to connect to a Postgres database in your local environment. The application starts emitting errors in the logs that look like this:

```
psql: error: could not translate host name "db.smith.lab" to address: Temporary failure in name
resolution
```

It appears that the application can't resolve the DNS record for *db.smith.lab*. There can be multiple reasons for the failure in name resolution. We'll explore a few tools to help troubleshoot this error. Before that, though, you really need to understand how your host uses DNS.

resolv.conf

The first place to start investigating DNS issues on any Linux host is the */etc/resolv.conf* file that provides information on what DNS servers to query and any special options needed (like timeout or security). The following is a *resolv.conf* file from a typical Ubuntu host:

```
# This file is managed by man:systemd-resolved(8). Do not edit.
#
# This is a dynamic resolv.conf file for connecting local clients to the
# internal DNS stub resolver of systemd-resolved. This file lists all
# configured search domains.
#
# Run "resolvectl status" to see details about the uplink DNS servers
# currently in use.
#
# Third party programs must not access this file directly, but only through
# the symlink at /etc/resolv.conf. To manage man:resolv.conf(5) in a
# different way, replace this symlink by a static file or a different
# symlink.
#
# See man:systemd-resolved.service(8) for details about the supported
# modes of operation for /etc/resolv.conf.

nameserver 127.0.0.53
options edns0 trust-ad
```

The file contains several comments describing `systemd-resolved`, and most importantly, it notes that you shouldn't edit it. This file is controlled by the `systemd-resolved` service provided by *systemd*, and it will overwrite the file next time the host or service restarts. After the comments, the second line from the bottom contains the `nameserver` keyword and the IP address

of the DNS server to query. On this Ubuntu host, the `nameserver` is set to `127.0.0.53`, which means any DNS requests will be sent to this address. If the local `resolver` does not know the answer to the query, the `resolver` will forward the request to an upstream `DNS server`.

The DNS upstream servers are usually set when you receive an IP address lease from a DHCP server. These upstream DNS servers can be internal servers that handle all your requests, or they can be any of the many public servers that the internet uses. For example, Cloudflare hosts public DNS servers at 1.1.1.1. There are quite a few public DNS servers around the globe.

The last line in the file modifies some specific resolver attributes using the `options` keyword. In this example, the `edns0` and `trust-ad` options are set. The `edns0` option enables expanded features to the DNS protocol. See RFC 2671 (*https://tools.ietf.org/html/rfc2671/*) for more details. The `trust-ad`, or authenticated data (AD) bit, option will include the authenticated data on all outbound DNS queries and preserve the authenticated data in the response. This will allow the client and server to validate the exchange between each other. This option is a part of a larger set of extensions that add security to DNS. See *https://www.dnssec.net/* for more information.

resolvectl

In this example host's *resolv.conf*, the DNS server is set to `127.0.0.53`, which is a local resolver that proxies any DNS request it does not know about. Each DNS server typically will have an upstream server that it forwards unknown requests to. Since you are using `systemd-resolver`, you can use a tool called `resolvectl` to interact with your local resolver. If this command line application is missing, you can install it via your package manager.

You'll want to know where your local DNS resolver (`127.0.0.53`) sends unknown requests. This might help you figure out why *db.smith.lab* resolution is failing. To see what DNS servers the resolver points to upstream, enter the following command:

```
$ resolvectl dns
Global:
--snip--
Link 2 (enp0s3): 10.0.2.3
```

The results show the downstream DNS server is set to `10.0.2.3` for interface `enp0s3`, which is the default interface and route on this host. Your setup and interface might be different. When any application on this host tries to connect to *db.smith.lab*, it first sends a DNS request to `127.0.0.53`, asking what IP address the hostname resolves to. The local resolver first looks for the answer locally. If the mapping is there, the results are returned immediately. However, if the answer is unknown, the resolver forwards the request to the upstream DNS server at IP `10.0.2.3`. Now, if the DNS server at `10.0.2.3` knows the answer for *db.smith.lab*, it will return a response to the local resolver, which in turn will respond to the user. If it doesn't know the

answer, the upstream server will forward that request to its upstream server until it reaches the authoritative server for the domain it's looking for.

Now that you know the IP address of your local resolver and upstream DNS server, you can query both to look for clues.

dig

The `dig` command line tool queries DNS servers and displays the results. This is extremely handy when you are troubleshooting DNS issues or need to fetch an IP address for a host. All you need to do is pass `dig` the hostname, and the response will provide information about the query and server that is responding.

Try querying the local resolver for the IP address of *db.smith.lab*. Enter the following command:

```
$ dig db.smith.lab
;  DiG 9.16.1-Ubuntu  db.smith.lab
;; global options: +cmd
;; Got answer:
;; -HEADER- opcode: QUERY, status: ❶SERVFAIL, id: 35816
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;db.smith.lab.                  IN      ❷A

;; Query time: 32 msec
;; SERVER: ❸127.0.0.53#53(127.0.0.53)
--snip--
```

The `status` field ❶ lets us know whether the query was successful. A successful query would have a status of `NOERROR`. In this example, the status is set to `SERVFAIL`, showing that no answer could be given. This makes sense, as the local DNS does not know where to find *db.smith.lab*. The `QUESTION SECTION` displays the query that was sent to the DNS server. In this case, the query is for the A record for *db.smith.lab* ❷. (An *A record* is a type of DNS record that maps a domain to an IP address.) The `SERVER` section tells us which DNS server was contacted to make the query. In this example, it's the local resolver (`127.0.0.53`) ❸, as expected.

To test your upstream server, you can instruct `dig` to talk to a specific DNS server instead of the local one. This will let you verify whether DNS resolution is failing locally or upstream. To do this, enter the following command:

```
$ dig @10.0.2.3 db.smith.lab
...
;; -HEADER- opcode: QUERY, status: SERVFAIL, id: 57409
...
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
```

```
;; QUESTION SECTION:
;db.smith.lab.                  IN      A
...
;; Query time: 32 msec
;; SERVER: 10.0.2.3#53(10.0.2.3)
;; WHEN: Sat Jun 19 18:20:23 UTC 2022
;; MSG SIZE  rcvd: 116
```

The `@10.0.2.3` parameter makes `dig` skip the local DNS and query the upstream host directly. The results, however, are the same, and you received a `SERVFAIL` for the status. This means the upstream server couldn't provide an answer for the hostname. You know you queried the correct server, because the `SERVER` section now states `10.0.2.3` instead of `127.0.0.53`.

NOTE *Pass `dig` the `+short` flag to show only the IP address, if it exists.*

To be safe, you should try one more query to make sure the local and upstream DNS servers are working correctly. First, you'll query for a DNS record that you are positive will return a response. This will let you verify whether DNS is broken for any domains, not just *db.smith.lab*. Enter the following command to query the A record for google.com:

```
$ dig google.com
...
;; -HEADER- opcode: QUERY, status: NOERROR, id: 15154
...
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             300     IN      A       142.250.72.78

;; Query time: 36 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
...
```

The status is `NOERROR`, and you received the A record of `142.250.72.78` in the `ANSWER SECTION`. This means the DNS server can resolve another hostname without error, but for some reason, it doesn't know about the *db.smith.lab* A record. Note that when there is an error or no answer to be given, the `ANSWER SECTION` is omitted from the results.

Next Steps

If there are resolution issues with a given hostname and DNS is functioning correctly and can resolve other hostnames, then the issues might stem from a DNS resolver that is missing the information that maps the hostname to an IP address. If your DNS is hosted on a service like Amazon Route53, make sure the record has not been removed by configuration management software or due to human error. If you manage the DNS server locally, you

can look to see if the A record is present. If it is not, perhaps the configuration contains some syntax error preventing the record from being served, or perhaps the DNS server needs to be restarted to read in its new records.

Scenario: Out of Disk Space

You will run out of disk space eventually. When this happens, you need to find out what is using all the space. The culprit could be anything from a misbehaving application to uncapped logfiles to a buildup of Docker images. To find the source of the problem, you'll first need to figure out which drive and filesystem are low on space. Once you locate those pieces, you will be able to search for files on the disk that may be using a lot of space.

df

The `df` command displays the free disk space on all the mounted filesystems on a host. It has multiple options, but the `-h` flag (for human-readable) is probably all you'll need. To see the free space on the mounted filesystems, enter the following command in a terminal:

```
$ df -h
Filesystem     Size  Used Avail Use% Mounted on
/dev/vda1       25G   25G     0 100% /
--snip--
```

In this example, device */dev/vda1* is using `100%` of its `25G` of disk space. The filesystem is mounted at */*, which is the root directory. If your host has multiple mounted disks, they'll be visible in the output as well.

find

The `find` command searches the filesystem for directories and files, and you can filter it to narrow down the search by looking for files that match only certain criteria or a specific directory. You can also locate files by their sizes on disk.

In your example, since you know the *root* filesystem is out of space after running the `df` command, you should direct `find` to search there. You'll execute the `find` command and search the *root* filesystem, looking for any files over `100M`. You'll sort them by size and display the top 10 with the `head` command. This could take a while, depending on the number of files on your drive. Enter the following command:

```
$ sudo find / -type f -size +100M -exec du -ah {} + | sort -hr | head
--snip--
10G   /var/log/php7.2-fpm.log
5G    /var/lib/docker/containers/.../...a3b76-json.log
--snip--
```

For each file located that is more than 100M, you'll execute (`-exec` flag) the `du -ah` command to fetch the file size on disk in human-readable format. The results, with file size, are sorted with largest files first. Then, the first 10 results are displayed.

This output shows a file named *php7.2-fpm.log* that is located under */var/log* and is `10G` in size. Also, a Docker container log located in */var/lib/docker/containers* is using `5G` of space. Together, these files are taking up 15GB of space on your disk. Usually, application logs like these should rotate and not become so large. The fact that both files are so big should trip your Spidey sense that something is not right here.

With more breadcrumbs to follow, check to see what process, if any, is using the *php7.2-fpm.log* file before you form a hypothesis.

lsof

Use the `lsof` command to list open files on a host. Files on a Linux host can be regular files, directories, or sockets, to name just a few. You can search for files owned by a particular process or by a specific user.

You'll use `lsof`, which requires elevated privileges, to find the process writing to the */var/log/php7.2-fpm.log* file. Enter the following command:

```
$ sudo lsof /var/log/php7.2-fpm.log
  COMMAND   PID  USER  FD  TYPE DEVICE    SIZE/OFF  NODE  NAME
php-fpm7. 23496  root  2w   REG  252,1  1048580000  1529  /var/log/php7.2-fpm.log
--snip--
```

You must pass the full path to the file you are interested in. In this case, it's the logfile. The `php-fpm7` command with the `PID 23496` owns the logfile in question. The file descriptor is `2w`, which means the file's descriptor is `2` and the file is opened for write access (`w`). The `TYPE` of file is `REG` (regular), representing a typical ASCII text file.

Next Steps

When your free disk space is low and you have tracked down a file that is contributing to the lack of space, you have a couple of options to remedy the situation. Since this logfile is currently being used, truncating or deleting it out from under the `php-fpm7` process isn't wise. Doing so could cause the process to die or stop writing logs completely. Instead, you can start by looking at the log output to see whether there are any telling errors or the application log level is perhaps stuck on `debug`. Also, there might be some correlation between this logfile and the fact that a Docker container log is large. Perhaps this process is running inside that container. Check the contents of the container log as well for any visible errors. On a housecleaning note, you should always make sure the host is set up to use the `logrotate` command to compress and rotate logfiles on a schedule. This can keep your logfiles from growing unbound and eating up your disk space. The `logrotate` configuration files are located in the */etc/logrotate.d/* directory on Ubuntu systems.

Scenario: Connection Refused

Sometimes, services refuse connections and do not leave an obvious reason why. For example, say you have an internal API that is reporting a high error rate, and say other services that use this API are throwing a lot of errors as well. The errors in the application logs would look something like this:

```
Failed to connect to api.smith.lab port 8080: Connection refused
```

It appears users are receiving a `Connection refused` error when trying to connect to the API server. You know the Docker container is up and running, or you would have gotten an alert that it was down. To troubleshoot this, you'll use a few commands that will help you identify any network- or configuration-related issues.

curl

Anytime you need to check whether a web server is responding to requests or just want to fetch some data or a file, turn to the `curl` command. For this example, you'll want to verify that an endpoint is down for everyone and that there is not just a routing issue on other hosts. The API server should respond with an `HTTP 200` status if it is functioning properly. To double-check that the API server is refusing connections, you could use `curl` by entering the following command:

```
$ curl http://api.smith.lab:8080
curl: (7) Failed to connect to api.smith.lab port 8080: Connection refused
```

The output shows you are getting a `Connection refused` error as well. This usually means the host is not listening on your port or a firewall is rejecting packets. Regardless of the reason, something is breaking your API requests.

NOTE *Another common connection error you will encounter is `connection timeout`. This error occurs when there is nothing responding to the request or a firewall is silently dropping the packets.*

ss

The `ss` (socket statistics) command is used to dump socket information on a host. For your troubleshooting scenario, you'll use it to see whether any application on the host is bound (or listening) to requests on `port 8080`. Enter the following command:

```
$ sudo ss -l -n -p | grep 8080
... 0.0.0.0:8080 0.0.0.0:* users:(("docker-proxy",pid=1448197,fd=4))
--snip--
```

The `-l` flag shows all the listening sockets on the host. The `-n` flag instructs `ss` not to resolve any service names like HTTP or SSH, and the `-p` flag shows the process that's using the socket. For `ss` to determine which

process owns the socket, `sudo` or elevated permissions are required. I truncated the beginning of the output line for readability, but the important part shows that the `docker-proxy` process is listening on all interfaces for port 8080 (`0.0.0.0:8080`). Next, you can verify that the requests destined for *api.smith.lab* are making it all the way to the host, where it lives.

NOTE *Before `ss`, there was a tool called `netstat`. The two tools basically do the same thing, but `netstat` is considered obsolete by today's standards. Most likely, you will still see tutorials and blog posts that still use `netstat`. Nevertheless, you should use `ss` going forward.*

tcpdump

One way to verify network traffic on a host is with the `tcpdump` command, which has many options and can capture traffic on one or all interfaces. It can even write out the network capture into a file for later analysis. Not only is `tcpdump` great for troubleshooting network issues, but you can use it for security auditing as well. For your example, you'll use it to capture network traffic intended for the *api.smith.lab* host on port 8080. This will let you know whether traffic being sent to that host is reaching its target, and it will hopefully shed some light on why you are getting the `Connection refused` error message.

On the host where the API application is running, enter the following command in a terminal. This will start the network packet capture on all interfaces for any TCP packet headed for port 8080 (note that elevated privileges are needed to listen on a network interface):

```
$ sudo tcpdump -ni any tcp port 8080
IP 192.168.50.26.50563 > 192.168.50.4.8080: Flags [S], seq 3446688967, win 65535, options [mss
1460,nop,wscale 6,nop,nop,TS val 157893401 ecr 0,sackOK,eol], length 0
IP 192.168.50.4.8080 > 192.168.50.26.50563: Flags [R.], seq 0, ack 3446688968, win 0, length 0
IP 192.168.50.26.50563 > 192.168.50.4.8080: Flags [S], seq 3446688967, win 65535, options [mss
1460,nop,wscale 6,nop,nop,TS val 157893501 ecr 0,sackOK,eol], length 0
IP 192.168.50.4.8080 > 192.168.50.26.50563: Flags [R.], seq 0, ack 1, win 0, length 0
```

The `-n` flag makes sure you do not try to resolve any host or port names. The `-i` flag tells `tcpdump` the network interface on which to listen. In this case, the term `any` is specified and means "Listen on all interfaces." You want to capture all packets destined for port 8080 since there might be numerous network interfaces on this host. The final `tcp port 8080` parameter states that you want only TCP packets that have port 8080 in them. These will include packets from both the client and the server.

Let's focus on the parts of the output that help with the `Connection refused` error problem. On the first line, the `IP` section shows that something from source `IP 192.168.50.26` is trying to connect to `192.168.50.4` on port `8080`. The `>` (greater-than) sign tells us the direction of the communication from one IP to another. The `Flags` being set show the types of network packets being sent. The first packet has an `S` (synchronize) flag. Anytime a client wants to establish a connection to another host, it sends the synchronize packet. In the next packet, host `192.168.50.4` responds to `192.168.50.26` with

a reset (R) packet. A reset packet is usually sent when there is an unrecoverable error and the server wants the client to terminate the connection immediately. Undeterred by the "Get off my lawn!" reset packet, the client tries again with another synchronize packet, which in turn causes server 192.168.50.4 to send another reset packet back to 192.168.50.26. The client at 192.168.50.26 finally takes a hint, and the connection is closed.

The flags show this connection isn't normal. A normal TCP connection starts off with a SYN packet from the client, followed by a SYN-ACK packet from the server. Once that packet is received, the client sends back an ACK packet to the server, acknowledging the last packet. This is referred to as a *three-way handshake*. See Figure 10-2 for details.

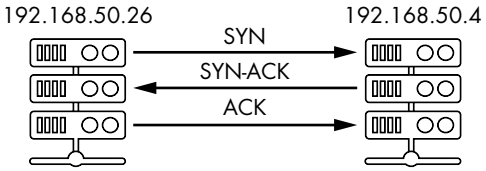


Figure 10-2: TCP three-way handshake

You clearly do not see any other packets (except resets) being sent from the server. The reset packets will cause the connecting clients to report that the connection is being refused. The good news is you verified that connections are making it all the way to server. The bad news is you still do not know why you are being refused.

NOTE *Visit* https://en.wikipedia.org/wiki/Transmission_Control_Protocol *for more information.*

Next Steps

At this point, you know the service is listening on port 8080. You verified this with the ss command. You also know traffic is making it all the way to the server, according to your network capture with tcpdump.

The next places to look are the Docker container and the application configuration. It is possible docker-proxy is having issues and not forwarding the traffic to the container running the API. Another possibility is that the container was started with incorrect internal port mappings. You know the external port, 8080, is mapped correctly, since it is listening for connections. But it's possible the mapped internal port is misconfigured. You can check both of these scenarios by looking at Docker's system logs for proxy errors, or by running **docker ps *<container id>*** or **docker inspect *<container_id>*** to check the port mappings.

Searching Logs

In almost every troubleshooting scenario, you'll most likely need to check logs. System and application logs hold a wealth of information you can view from the command line. Modern Linux distributions use systemd, which

has a log-collection mechanism called the *journal* that pulls in log events from multiple sources like *syslog*, *auth.log*, and *kern.log*. This lets you view and search logs in a single stream. As a troubleshooting archaeologist, you should know where logs are located and how to view and parse them.

Common Logs

Most system and application logs on a Linux host are stored in the */var/log* directory. The most common logs on a host that will aid in troubleshooting are *syslog*, *auth.log*, *kern.log*, and *dmesg*. Depending on your Linux distribution, the names of the logfiles may be different.

/var/log/syslog

The *syslog* file contains general global system messages for the Linux OS. Here is an example of a log line for `systemd`, stating that the logs are finished rotating:

```
Jun 11 00:00:03 box systemd[1]: Finished Rotate log files.
```

The line begins with a timestamp, followed by the host it is on (`box`) and the process (`systemd[1]`) that is reporting the log event. The last part of the line is the text message. This structured line format, also called *syslog*, is the default protocol for logging on a Linux host.

NOTE *The two most widely used versions of the* syslog *protocol are 3164 (*https://tools.ietf.org/html/rfc3164/*) and 5424 (*https://tools.ietf.org/html/rfc5424/*). Although some systems still use 3164's format, the 5424 format is the official standard of the* syslog *protocol.*

/var/log/auth.log

The *auth.log* file contains information regarding authorization and authentication events. This makes it a great place to investigate user logins and brute-force attacks, or to track a user's `sudo` commands. Here is an example of an *auth.log* message:

```
Jan 15 20:57:35 box sshd[27162]: Invalid user aiden from 192.168.1.133 port 59876
```

This message shows a failed login attempt over SSH for the user *aiden*, from the IP address `192.168.1.133`.

/var/log/kern.log

The *kern.log* is a good place to look for Linux kernel messages, such as hardware issues or general information related to the Linux kernel. The following log line shows the Linux out of memory manager (OOM) in action:

```
Jan 16 19:18:47 box kernel: [2397.472979] Out of memory: Killed process 20371 (nginx) total-
vm:571408kB, anon-rss:524540kB, file-rss:456kB, shmem-rss:8kB, UID:0 pgtables:1100kB oom_score
_adj:1000
```

Process 20371 was killed by the Out of memory manager because the system was running low on memory.

/var/log/dmesg

The *dmesg* log contains bootup messages from the host since last boot time. These messages can be anything from a USB device being recognized to a possible SYN packet flood attack. This sample log line from *dmesg* shows a Network driver being loaded into the kernel:

```
[1.036655] kernel: e1000: Intel(R) PRO/1000 Network Driver - version 7.3.21-k8-NAPI
```

The *dmesg* log has its own command line application, dmesg, to view the kernel ring buffer in real time. The dmesg command prints information, just like the *dmesg* log, but it can show information after bootup as well. You can also use it to troubleshoot multiple scenarios, such as port exhaustion, hardware failures, and OOM.

Common journalctl Commands

On a host that is using systemd, all of these common logs are stored in a single binary stream called a journal, which is orchestrated by the journald daemon. You can access the journal with the journalctl command line application. The journal is a handy troubleshooting tool because you can use it to view and search multiple logs at the same time. The journalctl command mimics many other logging commands you've discussed in this book, such as tail, minikube minikube kubectl -- logs and docker logs.

Say you want to review the logs, with the newest lines first. Enter the **sudo** command and pass the **-r** flag (reverse) to **journalctl** to view all logs in that order:

```
$ sudo journalctl -r
-- Logs begin at Sat 2022-02-27 23:10:19 UTC, end at Sun 2022-02-28 18:18:29 UTC. --
Feb 28 18:18:29 box sudo[73978]: pam_unix(sudo:session): session opened for user root by
vagrant(uid=0)
Feb 28 18:18:10 box systemd[7265]: Startup finished in 66ms.
--snip--
```

This output shows log lines for all services, with newest lines first.

Next, view logs during a certain time frame with the --since flag. Enter the following command:

```
$ sudo journalctl -r --since "2 hours ago"
-- Logs begin at Sat 2022-02-27 23:10:19 UTC, end at Sun 2022-02-28 18:27:20 UTC. --
Feb 28 18:27:20 box sudo[74471]: pam_unix(sudo:session): session opened for user root by
vagrant(uid=0)
Feb 28 18:27:20 box sudo[74471]:  vagrant : TTY=pts/2 ; PWD=/home/vagrant ; USER=root ;
COMMAND=/usr/bin/journalctl -r --since 2 hours ago
--snip--
```

This output shows the logs that have a timestamp starting `2 hours ago` up till the current time, when the command is run. With the `-r` flag, the newest logs are displayed first.

You can filter logs based on a `systemd` service name. For example, to view all the logs that were written by the SSH service, enter the following command to pass the `-u` (unit) flag to `journalctl`:

```
$ sudo journalctl -r -u ssh
--snip--
Feb 27 23:17:31 ... sshd[16481]: pam_unix(sshd:session): session opened for user akira by
(uid=0)
Feb 27 23:17:31 ... sshd[16481]: Accepted publickey for akira from 10.0.2.2 port 55468 ...
--snip--
```

The output shows log lines for SSH pertaining to a login `session`, in reverse order.

You can also display log lines that match a specific log level, like info or error. Choose the priority level (`-p`) by using keywords like `info`, `err`, `debug`, or `crit`. The following is the same command as above but with the `-p err` flag to show only error logs from the SSH daemon:

```
$ sudo journalctl -r -u ssh -p err
--snip--
Feb 28 08:39:13 box sshd[4182]: error: maximum authentication attempts exceeded for root from
192.168.25.4 port 34622 ssh2 [preauth]
--snip--
```

The output shows an `error` log line where the *root* user reached the maximum failed login attempts.

Narrowing down logs to a specific time frame or showing log lines that match a given log level is great, but what if you want to find a specific message in the journal stream? The pattern-matching flag (`-g`) in `journalctl` can match any message using a regular expression. The following example searches the SSH logs for the `session opened` message. Enter the following command:

```
$ sudo journalctl -r -u ssh -g "session opened"
--snip--
Jun 10 21:31:40 box sshd[2047134]: pam_unix(sshd:session): session opened for user vagrant by
(uid=0)
Jun 09 16:49:10 box sshd[2008012]: pam_unix(sshd:session): session opened for user x7b7 by
(uid=0)
--snip--
```

Here, SSH sessions for two different users (*vagrant* and *x7b7*) are filtered out.

WARNING *If you are using an older version of `journald`, the `grep` pattern matching might not be included. If this is the case, you can pipe the search results to the `grep` command by entering this command: **`sudo journalctl -r -u ssh | grep "session opened"`**.*

The journalctl tool is helpful when you want to view many logs at once, but you'll also encounter logs that are not captured in the journal system.

Parsing Logs

Parsing logs is a key troubleshooting skill. In addition to journalctl, you can parse and traverse logs with the grep and awk commands. The grep command is used to search for patterns in text or a file. The awk command is a scripting language tool that can filter text, but it also has more advanced features like built-in functions for math and time.

grep

The grep command allows you to search for a pattern quickly. For example, to use grep to find any occurrences of the IP address 10.0.2.33 in */var/log/syslog*, pass grep the search pattern and the file to search by entering this command:

```
$ grep "10.0.2.33" /var/log/syslog
... box postfix/smtpd[6520]: connect from unknown[10.0.2.33]
... box postfix/smtpd[6520]: disconnect from unknown[10.0.2.33] ehlo=1 auth=0/1 quit=1
commands=2/3
```

This command returned two log lines for the postfix daemon containing the 10.0.2.33 IP address.

To find users trying to execute the sudo command who don't have permission, search */var/log/auth.log* using grep by entering the following command:

```
$ grep "user NOT in sudoers" /var/log/auth.log
Jan 31 17:37:40 box sudo: akira : user NOT in sudoers ; TTY=pts/0 ; PWD=/home/akira ; USER=root
; COMMAND=/usr/bin/cat /etc/passwd
```

The search pattern "user NOT in sudoers" indicates an unauthorized sudo attempt violation. This search returns one match showing that the user *akira* tried to read the contents of the */etc/passwd* file but was denied.

Taking it one step further, it would be helpful to check the *auth.log* to see what else this user was doing around the same time. To get extra log lines with grep, use the -A flag to grab a given number of lines after the matched lines or use the -B flag to fetch a given number of lines before the matched results. You can also use the -C flag to fetch before and after the match, simultaneously.

Now, you should grab the five log lines before the log line alerting to the sudo violation for the user *akira*. This will help you get an idea of what else might have been going on around that time in the log. Enter the following command:

```
$ grep -B 5 "user NOT in sudoers" /var/log/auth.log
Jan 31 17:37:35 box sshd[64646]: pam_unix(sshd:session): session opened for user akira by
(uid=0) ❶
Jan 31 17:37:35 box systemd-logind[632]: New session 169 of user akira.
```

```
Jan 31 17:37:35 box systemd: pam_unix(systemd-user:session): session opened for user akira by
(uid=0)
Jan 31 17:37:38 box sudo: pam_unix(sudo:auth): Couldn't open /etc/securetty: No such file or
directory
Jan 31 17:37:40 box sudo: pam_unix(sudo:auth): Couldn't open /etc/securetty: No such file or
directory
Jan 31 17:37:40 box sudo: akira : user NOT in sudoers ; TTY=pts/0 ; PWD=/home/akira ; USER=root
; COMMAND=/usr/bin/cat /etc/passwd ❷
```

The first five lines show the user *akira* logging in over SSH ❶. Within five seconds of logging in (`17:37:35` to `17:37:40`), the user *akira* tried to read the contents of the */etc/passwd* file ❷. Without the extra context, it might be tempting to overlook this action, but after seeing the user's behavior upon logging in, grabbing additional lines around a match can provide more insight.

awk

The `awk` command can search for specific patterns like `grep` does, but it can also filter out information from any column. For this example, you should grab all the source IP addresses from the requests in */var/log/nginx/access.log*. This log contains all the requests to a website proxied by Nginx. The source IP address is usually the first column in the log line, unless you have modified Nginx's default logging format. You'll use `awk`'s `print` function and pass the `$1` argument so it prints only the first column. By default, `awk` splits columns on whitespace. Enter the following command:

```
$ sudo awk '{print $1}' /var/log/nginx/access.log
127.0.0.1
192.168.1.44
```

The output shows only two IP addresses. Clearly, it's not a busy web server, but the output doesn't show the whole log line as do the previous `grep` examples. You can parse the text and display the column of your choosing with the `awk` command. Each column in the log line is given a unique column number. For example, to see only the date timestamps (fourth column) in the *access.log*, pass `$4` to the `print` function. If you want to return more than the one column, pass multiple column numbers to the `print` function, separating each from the next with a comma, like this: **`'{print $1,$4}'`**.

You'll use `awk` to search for all the HTTP 500 response code, which is usually in the ninth column (`$9`) in the Nginx *access.log* file. Enter the following command:

```
$ sudo awk '($9 ~ /500/)' /var/log/nginx/access.log
10.0.2.15 - - [15/Feb/2022:19:41:46 +0000] "GET / HTTP/1.1" 500 396 "-" "Mozilla/5.0 (X11;
Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.120 Safari/537.36"
```

Inside the parentheses, the tilde (`~`) is a field number that tells `awk` to apply the search pattern only to a specific column. In this case, you want to search in the ninth column for anything matching 500. The command returned a single result for a `GET` request that responded with an HTTP `500`.

You can change the search pattern to suit your needs. For instance, if you want to search the logs for any unauthorized HTTP requests, change the pattern of `/500/` to `/401/`. To expand on this even further, you can change the search pattern from `/500/` to `/404/` and add a requirement that any 404 responses must be from an HTTP POST method. You do this by adding an `if` conditional block to `awk`. To search for any lines that match those criteria, enter the following in a terminal:

```
$ sudo awk '($9 ~ /404/) {if (/POST/) print}' /var/log/nginx/access.log
127.0.0.1 - - [31/Jan/2022:18:16:45 +0000] "POST /login HTTP/1.1" 404 162 "-" "curl/7.68.0"
```

The search pattern is like the previous one. Match the value at column `$9` to the number `404`. Then pass an `if` block that states, "If the line from the column `$9` match contains the word `POST` anywhere in it, print that whole log line." The result shows an HTTP `POST` to the */login* path that returned an HTTP `404`.

NOTE *You can also use OR (|) logic in your search pattern. For example, to search for a HTTP `401` or `403` error code, you could do something like this: `sudo awk '($9 ~ /401|403/) /var/log/nginx/access.log`. Notice how the pipe operator splits the values.*

Probing Processes

Sometimes, you won't encounter many symptoms when investigating issues on a host. The health stats may look okay, the logs may show nothing interesting . . . but something will still not be right. Maybe a scheduled job didn't execute cleanly, or an application appears to be hung. One way to dig deeper is to investigate the running process on the host.

strace

The `strace` command traces system calls and signals, allowing you to attach to a process and gain valuable knowledge in real time. Your application uses system calls to ask the Linux kernel to perform tasks like opening a network socket, reading and writing a file, or creating a child process. You should use the `strace` command to troubleshoot a process that looks for issues in these calls, or when you need an overview of what a process is doing. Note that the `strace` command needs *root* privileges since it is attaching to another process.

Many system calls are available, but here are a few for reference:

`open()` Create or open files.

`read()` Read from a file descriptor.

`write()` Write to a file.

`connect()` Open a connection.

`futex()` Wait or wake up threads when a condition becomes true (blocking lock).

WARNING *The `strace` command can be very verbose and may cause performance issues for the process you are probing. Use it with caution in a production environment.*

Now, you should trace a process. The following command attaches to the running process `19419`, which is the Greeting web server from Chapter 4 and prints out any system calls that are happening when the trace begins:

```
$ sudo strace -s 128 -p 19419
strace: Process 19419 attached
--snip--
accept4(5, {sa_family=AF_INET, sin_port=htons(64221), sin_addr=inet_addr("172.28.128.1")},
[16], SOCK_CLOEXEC) = 9
--snip--
recvfrom(9, "GET / HTTP/1.1\r\nHost: 172.28.128"..., 8192, 0, NULL, NULL) = 82
getpeername(9, {sa_family=AF_INET, sin_port=htons(64221), sin_addr=inet_addr("172.28.128.1")},
[16]) = 0
--snip--
sendto(9, "HTTP/1.1 200 OK\r\nServer: gunicorn/20.0.4\r\nDate: Mon, 01 Feb 2022 22:03:12 GMT\r\
nConnection: close\r\nContent-Type: text/html; chars"..., 160, 0, NULL, 0) = 160
sendto(9, "<h1 style='color:green'>Greetings!</h1>", 39, 0, NULL, 0) = 39
--snip--
write(1, "172.28.128.1 - - [01/Feb/2022:21"..., 88) = 88
close(9) = 0
--snip--
```

The `-s` flag sets the message output size of `128` bytes. The `-p` flag tells `strace` which PID to attach to (in this case, it's `19419`). I cherry-picked some system calls from the output to make it easier to follow. The `accept4` system call creates a new connection from IP address `172.28.128.1` and returns file descriptor `9`. The `recvfrom` system call receives an HTTP `GET` request from a socket with file descriptor `9`. The first `sendto` system call sends an HTTP header response from the web server back over the socket. The following `sendto` system call transmits the body of the HTTP `GET` response back to the socket as well. The `write` system call writes what appears to be a *syslog* line to file descriptor `1`. Finally, the `close` system call is executed, closing the previous socket file descriptor `9`, which closes the network connection. You have captured the transaction between an HTTP client and an HTTP server for a `GET` request.

Now, imagine you're trying to investigate an issue but are lacking context on a process. You have exhausted other means, like log spelunking and metric watching. Everything seems in order, but your application is still not behaving correctly. You can use the summary flag (`-c`) for `strace` to get an overview of what system calls the process is using. It will output a running count of what system calls are being executed, how long each one is taking, and any errors that those calls return. Once you run the command, it will pause in the foreground while it collects data, and it won't display the results until you press CTRL-C. The longer you let it run, the more data you will accumulate.

The `strace` command has numerous flags and options to use for tracing. You can use the follow (`-f`) flag to follow any new processes created (forked)

from the parent. You can use the syscall (`-e`) flag when you want to track only specific system calls. You can use the summarize (`-c`) flag when you want an overall view of the system calls, timings, and errors. Finally, the output (`-o`) flag can be extremely useful for storing the trace output to a file so you can review and parse it later.

For example, enter the following command to fetch a summary for process ID `28485`:

```
$ sudo strace -p 28485 -c
strace: Process 28485 attached
 % time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
❶ 49.47    0.000141          14        10           sendto
  13.68    0.000039           2        17           fchmod
  10.53    0.000030           6         5           close
   7.37    0.000021           3         6           select
   7.02    0.000020           4         5           write
❷  7.02    0.000020           1        11         6 openat
   2.11    0.000006           1         6           getppid
   1.75    0.000005           0        10           getpid
   0.35    0.000001           0         5           ioctl
   0.35    0.000001           0         5           recvfrom
❸  0.35    0.000001           0        50           getpeername
   0.00    0.000000           0        10           getsockname
   0.00    0.000000           0        10           fcntl
------ ----------- ----------- --------- --------- ----------------
 100.00    0.000285                   150         6 total
```

The `% time` column shows the percentage of time each call made up during the trace capture. In this example, the process spent most of its trace time ❶ (before the trace was stopped) in the `sendto` system call. The `calls` column shows how many times the system call was executed. In this case, `getpeername` ❸ was executed the most (50 times). The `getpeername` call returns the IP address of the peer connected over the socket. During the trace, process `28485` counted six errors ❷ when calling the `openat` system call. You can use this call to open a file by its specified path name.

You should run `strace` again to focus on the errors for the `openat` system call. Enter the following command:

```
 $ sudo strace -p 28485 -e openat
--snip--
openat(AT_FDCWD, "/var/log/telnet-server.log", O_RDONLY) = -1 ENOENT (No such file or
directory)
--snip--
```

The output shows that process `28485` is trying to open the */var/log/telnet-server.log* file. The call is returning `-1`, which means the file does not exist. This matches the error output from the earlier summary. As you can see, being able to peer down into a running process and understand what it is doing at the system call level can be invaluable.

NOTE *Other tools can explore a process. The* `ltrace` *command is like* `strace`*, but it reports on the dynamic library calls made. The* `dtrace` *framework is also like* `strace`*, but it can trace kernel-level issues as well.*

Summary

Most of the scenarios described here reflect issues you will encounter throughout your career. Experience and repetition will help you build muscle memory for making quick work of these issues. My goal in describing these scenarios has been to show you how to use deductive reasoning to follow clues to find causes.

In this chapter, you learned about helpful forensic tools like `top`, `lsof`, `tcpdump`, `iostat`, and `vmstat`, which will help you diagnose symptoms. You also learned how to parse common logfiles using tools like `journalctl`, `grep`, and `awk`. All the tools and tactics discussed here should aid you the next time you find yourself trying to investigate problems.

This concludes Part III, which has been on monitoring and troubleshooting. You now can monitor and alert on any application you deploy to Kubernetes. You have also gotten a troubleshooting primer to help you investigate common problems that arise when managing hosts and software.

INDEX

Symbols and Numbers

A

B

H

I

J

K

L

M

N

O

T

U

V

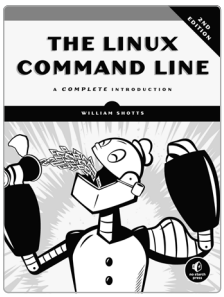
W

Y

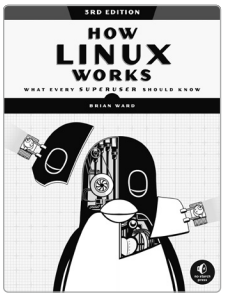
RESOURCES

Visit *https://nostarch.com/devops-desperate/* for errata and more information.

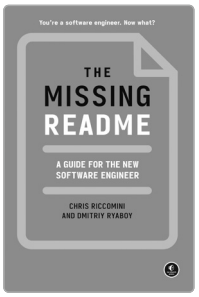
More no-nonsense books from **NO STARCH PRESS**



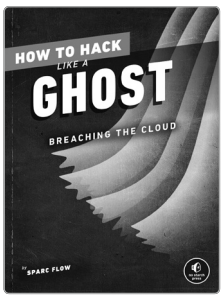
THE LINUX COMMAND LINE, 2ND EDITION
A Complete Introduction
BY WILLIAM SHOTTS
504 PP., $39.95
ISBN 978-1-59327-952-3



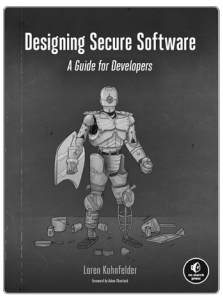
HOW LINUX WORKS, 3RD EDITION
What Every Superuser Should Know
BY BRIAN WARD
464 PP., $49.99
ISBN 978-1-7185-0040-2



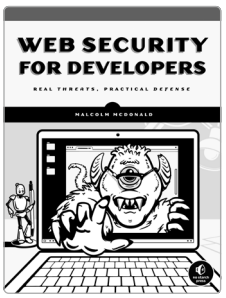
THE MISSING README
A Guide for the New Software Engineer
BY CHRIS RICCOMINI AND DMITRIY RYABOY
288 PP., $24.99
ISBN 978-1-7185-0183-6



HOW TO HACK LIKE A GHOST
Breaching the Cloud
BY SPARC FLOW
264 PP., $34.99
ISBN 978-1-7185-0126-3



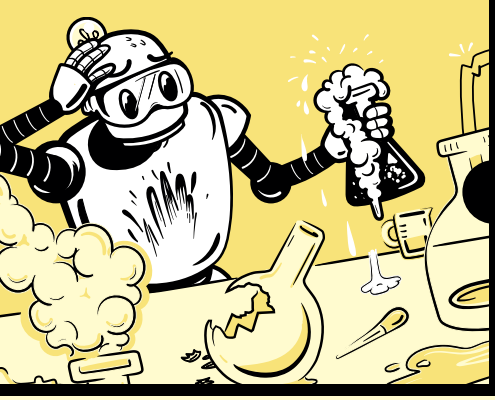
DESIGNING SECURE SOFTWARE
A Guide for Developers
BY LOREN KOHNFELDER
312 PP., $49.99
ISBN 978-1-7185-0192-8



WEB SECURITY FOR DEVELOPERS
Real Threats, Practical Defense
BY MALCOLM MCDONALD
216 PP., $29.95
ISBN 978-1-59327-994-3

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM
WEB:
WWW.NOSTARCH.COM



DevOps Basics for Engineers and Admins in Crisis Mode

If you're a software engineer, developer, or sys admin who needs to get up to speed with DevOps quickly, this book covers the basics you need to thrive in a modern application stack.

This book's fast-paced, hands-on examples will provide the foundation you need to start performing common DevOps tasks. You'll explore how to implement Infrastructure as Code (IaC) and configuration management (CM)—essential practices for designing secure and stable systems. You'll take a tour of containerization and set up an automated continuous delivery (CI/CD) pipeline that builds, tests, and deploys code. You'll dig into how to detect a system's state and alert on it when things go sideways.

You'll learn how to:

- Create and provision an Ubuntu VM with Vagrant and Ansible
- Manage users, groups, and password security
- Set up public key and two-factor authentication over SSH
- Automate and test a host-based firewall
- Use Docker to containerize applications and Kubernetes for orchestration
- Build a monitoring stack and troubleshoot problems and performance issues

DevOps for the Desperate is a practical, no-nonsense guide to get you up and running quickly in today's full-stack infrastructure.

ABOUT THE AUTHOR

Bradley Smith has been a DevOps and software engineer for more than 20 years at many startups, local governments, and businesses of varying sizes. He's solved countless technical challenges during his career, and he's built and trained many DevOps, SRE, and software engineering teams. He graduated from the University of Massachusetts Lowell and now resides in Denver, Colorado.

Coverage includes Ansible, Docker, Kubernetes, and more...

no starch press®

THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

$29.99 ($39.99 CDN)

ISBN 978-1-7185-0248-2

52999

9 781718 502482