

Compliments of



Road to Kubernetes

Justin Mitchel





Fast and simple Kubernetes cluster deployments

TRY NOW



Road to Kubernetes

Road to Kubernetes

JUSTIN MITCHEL



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

© 2024 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Connor O'Brien
Technical editor: Billy Yuen
Production Editor: Aleksandar Dragosavljević
Copy Editor: Katie Petito
Typesetter: Bojan Stojanović
Cover Designer: Marija Tudor

ISBN: 9781633436770
Printed in the United States of America

*To my wife Emilee—thank you for your endless support and encouragement.
I am a very lucky man to have you in my life. Thank you for giving me everything.
To my daughters McKenna, Dakota, and Emerson—thank you for being you.
Each one of you is exactly what our family needs, and I am so proud of the women you are
becoming. Being your Dad has been the greatest and most rewarding gift in the world.
I love each of you more than you can possibly know, and I am very excited for many more
adventures together. Love you all!*

brief contents

1	■	Kubernetes and the path to modern deployment	1
2	■	Creating the Python and JavaScript web apps	9
3	■	Manual deployment with virtual machines	37
4	■	Deploying with GitHub Actions	71
5	■	Containerizing applications	103
6	■	Containers in action	126
7	■	Deploying containerized applications	148
8	■	Managed Kubernetes Deployment	172
9	■	Alternative orchestration tools	223
A	■	Installing Python on macOS and Windows	249
B	■	Installing Node.js on macOS and Windows	258
C	■	Setting up SSH keys for password-less server entry	261
D	■	Installing and using ngrok	269

contents

preface xii
acknowledgments xv
about this book xvi
about the author xxi
about the cover illustration xxii

1 ***Kubernetes and the path to modern deployment*** 1

- 1.1 Anyone can deploy 2
- 1.2 Our path to deployment 3
- 1.3 The challenge of dependencies 4
- 1.4 What do containers do? 5
- 1.5 The challenges in building and running containers
without Kubernetes 6
- 1.6 What are the fundamental challenges with using
Kubernetes? 7
- 1.7 How does this book teach the technology? 8

2 ***Creating the Python and JavaScript web apps*** 9

- 2.1 Designing a basic FastAPI web app in Python 10
 - Python project setup with virtual environments* 10
 - Hello World with FastAPI* 13 ■ *Handling multiple routes with FastAPI* 14

- 2.2 Creating a JavaScript web application with Node.js and Express.js 15
 - Hello World from Express.js* 17 ■ *Run the Node.js + Express.js app* 19 ■ *Create a JSON response with Express.js* 20
- 2.3 Tracking code changes with Git 21
 - Using Git in our projects* 23 ■ *Ignoring files* 25 ■ *Tracking files* 26 ■ *Complete Git repo reboot* 29 ■ *Git logs and show* 30 ■ *Fundamental Git commands overview* 31
- 2.4 Pushing code to GitHub 32
 - Creating a GitHub account* 32 ■ *Create a Git repository on GitHub* 33 ■ *Configuring our local git repo for GitHub* 34 ■ *Rinse and repeat* 35

3 **Manual deployment with virtual machines** 37

- 3.1 Creating and connecting to a remote server 38
 - Provisioning a virtual machine* 39 ■ *Connecting via SSH* 40
- 3.2 Serving static websites with NGINX 43
 - Embrace ephemeral VMs* 44 ■ *Modify the default NGINX HTML page* 45
- 3.3 Self-hosted remote Git repositories 47
 - Log in to the server with SSH* 48 ■ *Creating the bare Git repositories* 48 ■ *Pushing local code to the remote repo* 50 ■ *Git hook to check out code* 51
- 3.4 Installing the apps' dependencies 53
 - Installing Python 3 and our FastAPI project* 54 ■ *Installing Node.js and our Express.js app* 58 ■ *Running the Express.js application* 61
- 3.5 Run multiple applications in the background with Supervisor 62
 - Installing Supervisor* 63 ■ *Configure Supervisor for apps* 63
- 3.6 Serve multiple applications with NGINX and a firewall 66
 - Configuring NGINX as a reverse proxy* 67 ■ *Installing Uncomplicated Firewall* 68

4 *Deploying with GitHub Actions* 71

4.1 Getting started with CI/CD pipelines with GitHub Actions 72

Your first GitHub Actions workflow 73 ■ *Creating your first GitHub Actions secret* 78 ■ *Installing the public SSH key on Akamai Connected Cloud* 79 ■ *Creating a new virtual machine on Akamai Linode* 81 ■ *Installing NGINX on a remote server with a GitHub Actions workflow* 82

4.2 Virtual machine automation with Ansible 85

GitHub Actions workflow for Ansible 86 ■ *Creating your first Ansible Playbook* 88 ■ *Ansible for our Python app* 90
Ansible for our Node.js app 97

5 *Containerizing applications* 103

5.1 Hello World with Docker 104

Running your first container 105 ■ *Exposing ports* 106
Entering a container 107 ■ *Stateless containers* 107

5.2 Designing new container images 108

Creating a Dockerfile 109 ■ *Running your first container image* 110

5.3 Containerizing Python applications 110

Version tags and a Python Dockerfile 110 ■ *Creating a Python Dockerfile* 113 ■ *Ignoring unwanted files in Docker builds* 115
Creating and running an entrypoint script 116

5.4 Containerizing Node.js applications 118

The Node.js entrypoint script and .dockerignore file 119
The Node.js Dockerfile 120

5.5 Pushing container images to Docker Hub 121

Your first push to Docker Hub 122

6 *Containers in action* 126

6.1 Building and pushing containers with GitHub Actions 127

Docker login and third-party GitHub Action tools 128
A GitHub Actions workflow for building containers 129

- 6.2 Managing a container with Docker Compose 130
 - Why we need Docker Compose 130* ■ *Docker Compose for our Python app 131* ■ *Docker Compose for our Node.js app 134*
- 6.3 Stateful containers with volumes 135
 - Using volumes with Docker 135* ■ *Using volumes with Docker Compose 137* ■ *Built-in volume management with Docker Compose 138*
- 6.4 Networking fundamentals in Docker Compose 140
 - Container-to-container communication simplified 141*
 - Databases and Docker Compose 143* ■ *Environment variables with dotenv in Docker and Docker Compose 145*

7 **Deploying containerized applications 148**

- 7.1 Hello prod, it's Docker 149
 - Provisioning a new virtual machine 149* ■ *Installing Docker on Ubuntu 150* ■ *Installing Docker Compose on Ubuntu 151*
 - Deploying NGINX 151*
- 7.2 Staging containers 153
 - Dockerfiles for multiple environments 154* ■ *Docker Compose for multiple environments 155*
- 7.3 GitHub Actions to deploy production containers 157
 - Building and hosting a production container 158*
 - Installing Docker and Docker Compose on a virtual machine with GitHub Actions 161* ■ *Using GitHub Actions to run Docker Compose in production 162*
- 7.4 The limitations of using Docker Compose for production 167
 - Scaling containers with Docker Compose 168*

8 **Managed Kubernetes Deployment 172**

- 8.1 Getting started with Kubernetes 174
 - Self-service or managed Kubernetes? 174* ■ *Provisioning a Kubernetes cluster 175* ■ *Core concepts and components 177*
- 8.2 Connecting to Kubernetes 179
 - The Kubernetes Dashboard GUI 179* ■ *Installing kubectl 181*
 - Configuring kubectl 183*

- 8.3 Deploy containers to Kubernetes 185
 - Your first Pod and manifest* 186 ■ *Your first Service* 187
 - From Pods to Deployments* 191 ■ *Customize NGINX with ConfigMaps* 193 ■ *Environment Variables with ConfigMaps and Secrets* 196
- 8.4 Volumes and stateful containers 200
 - Volumes and Deployments* 201 ■ *StatefulSets* 203
 - Container-to-container communication within Kubernetes* 208
 - Namespaces to manage cluster resources* 211
- 8.5 Deploy apps to production with Kubernetes 214
 - LoadBalancer Services* 215 ■ *Deploying to Kubernetes with GitHub Actions* 219

9 ***Alternative orchestration tools*** 223

- 9.1 Container orchestration with Docker Swarm 224
 - Preparing the Docker Swarm Manager* 226 ■ *Docker Compose for Docker Swarm* 229 ■ *Start Docker Swarm* 230 ■ *Deploy a Docker Swarm Stack* 231
- 9.2 HashiCorp Nomad 232
 - Preparing our Nomad cluster* 233 ■ *Installing Nomad* 234 ■ *Configuring the Nomad server* 235 ■ *Configuring the Nomad Clients* 238 ■ *Running containers with Nomad jobs* 240

- appendix A Installing Python on macOS and Windows* 249
- appendix B Installing Node.js on macOS and Windows* 258
- appendix C Setting up SSH keys for password-less server entry* 261
- appendix D Installing and using ngrok* 269
- index* 271

preface

I believe that more applications are going to be released in the next decade than in the previous decade to the 10th power. This exponential growth is going to be driven by modern deployment technologies like containers and container orchestration, along with the explosion in the use of generative AI tools. While the tools and technologies might change, I am certain that more applications will be deployed.

I have seen software go from floppy disks to compact disks (CDs), to cartridges (for game consoles), to DVDs, and ultimately to the internet. As the format of distribution changed, so did the number of applications that were released; they increased exponentially. Even if each physical format change did not see this exponential growth, the internet definitely did. I grew up in a time when I never had to write a single line of code that required a disc or cartridge for the app to work. My programming and web development skills were developed on the internet.

When I was 16, I decided I was going to make my fortune selling t-shirts. A friend agreed this plan was a great idea, so we joined forces and did what any smart teenager would: borrow money from our parents. While this money was enough to print shirts, it was not enough to build a website, or so we thought. At the time, MySpace was the most popular social media site, and eCommerce was just starting to explode. We wanted to look as legit as possible to potential retailers, so we figured creating a website was a great idea. The problem? Websites were at least \$800 to hire a company to create one. Because we spent all of our parents' money, we needed to do this one on our own.

That's when I met Bobby Kim, the founder of an up-and-coming clothing company, The Hundreds, and he gave me the advice I will never forget: "Oh man, you don't need to hire a website design company, buy a book and do it yourself! That's what we did; it's really not that hard." Despite not fully agreeing with the sentiment of "It's not that hard," I did buy the book. To my surprise, creating a basic HTML website was easy, thanks to the practical, hands-on nature of the book. While the website I created was definitely amateur by any standard, I was able to publish it on the internet with the help of a family friend. Achievement unlocked, and \$800 saved.

It might not be surprising to learn that a couple of 16-year-olds failed to compete with the likes of Quiksilver and Billabong right before a massive global recession. While that sentence might echo the current economic times and maybe would make a good one-liner for a movie, I hold that time period near and dear because that t-shirt company propelled me into the wonderful world of building things for the internet.

Fast forward a few years, and I found myself at odds with creating real software. While HTML might look like "real software," I consider it a bit more like a complicated text document (ahem, because it is). Once again, I was misguided in thinking that writing code was far too complex and thus not for me. Thankfully, during yet another one of my many projects, my technical cofounder told me I should try out this tool called Python. And so I did. I bet you know where the story goes from here.

It felt like it was a matter of months, and I was regularly releasing my Python-based web applications to the world faster than I ever could have hoped. These skills led to many freelancing opportunities and many more opportunities to learn and grow. That's exactly what I did.

Over time, I came to realize that I needed to help others overcome their own self-doubt in writing, learning, and releasing software to the world. A few months after this realization, a friend invited me to a weekend workshop called Startup Weekend, where various entrepreneurs and coders would come together and create businesses in just two days. I decided that instead of driving 10+ hours to a workshop in another state, I would just buckle down and create my first course. After a caffeine-filled weekend, I had my first four-hour course done and ready to release.

Here lies another inflection point on my journey to you reading these words. *Was I ready? Was the course any good? Will people find out that I am bad at this?* The questions went on and on in my head causing me to rethink my course altogether. During this time, two quotes rang in my head:

“Business is just two things: innovation and marketing.”

—Peter Drucker

“Just Do It”

—Nike

So, despite my self-doubt and self-defeating thoughts, I decided to release the course, which took over two weeks after creating it. I decided to name the course “Coding for Entrepreneurs” because it’s exactly who I am and who I identify with: a coding entrepreneur. I also figured that if the 16-year-old version of me saw the course “Coding for Entrepreneurs,” he would have purchased it in 2 seconds.

The course took off and quickly amassed 10s of thousands of students, which led to a successful Kickstarter campaign and eventually to hundreds of thousands of students across the world through platforms like Udemy, YouTube, CodingForEntrepreneurs.com, and many others. To date, I have more than 800,000 students enrolled on Udemy, more than 235,000 subscribers on YouTube, and more than 15,000 followers on GitHub.

I created this book as a result of the many journeys I have been on with software and, more importantly, deploying software. I believe that deployed software is the *only software* that will propel you or your team forward. I hear statements all the time about what the best place to deploy software is, and in all honesty, I do not think there will ever be a best. *Best* implies finished, done, over, and so on, whereas *better* implies there’s a chance for worse. At one point, file transfer protocol (FTP) was considered the best option for deploying code, and spoiler alert, it no longer is. While I just bemoaned the term *best*, I will say that I think containers and container orchestration are, as of the end of 2023, the best ways to deploy code into production. I suspect these tools will eventually change for something better.

The *Road to Kubernetes* is about learning various options to deploy your applications into production so that you can continuously do so. I wrote this book as a step-by-step journey so you can have practical and tangible results each step of the way to help you deploy more often and, if I did my job correctly, deploy better. The more you can deploy your software, the more momentum your project may have and thus end up helping others in the process. The more you can deploy software, the better it can get.

acknowledgments

First, I would like to thank you, dear reader. I created this book for you, and I hope you enjoy it. Please feel free to reach out to share any insights you may have.

I would like to thank my students and followers for all of your watching, reading, listening, sharing, and doing the work; I would not be where I am today if it weren't for you. Thank you for letting me be a voice in this group of outstanding people.

I would like to thank Akamai for their support in my educational and technological pursuits. Akamai has been a great partner in my mission to help others build software and release it to the world.

I would like to thank Timothy Ryan for his support in making this book possible and for being an ear for my long-winded discussions about all things technology. I would also like to thank Andrew, Justin, Jamie, Hillary, Kerterika, Maddie, Talia, and other members of the Akamai Connected Cloud team for your help on this and many other projects. While this list is not exhaustive, I do appreciate you and what you have done over the time we have worked together.

Last, I would like to thank Andy, Ian, and Connor at Manning for helping bring this project to life.

about this book

With *Road to Kubernetes*, we start with creating simple Python and Node.js web applications. These applications will stay with us for our entire journey so we can learn about all the various tools and techniques to deploy them. From there, we start using secure shells (SSH) and modern version control by way of Git with self-managed environments, self-managed repositories, and self-managed deployments.

After we understand the self-managed way of deploying, we move to automation with GitHub, GitHub Actions, and even Ansible. GitHub is a popular third-party-managed git repository and code hosting service that also allows us to run one-off code workflows called GitHub Actions. These workflows are short-lived computing environments that are useful to build, test, and deploy our code and help us to continuously integrate and deliver (CI/CD) our apps. Ansible helps us automatically configure our deployment environments (e.g., virtual machines) by declaring what software we need to run our applications.

While Ansible is great at configuring our environments after the fact, we started adopting a way to preconfigure our environments into portable and manageable runtime environments called containers. Containers and the process of containerization were pioneered by Docker and are often known as Docker Containers. These containers are essentially apps themselves that include a small operating system to run our code—think of it as a tiny Linux OS that runs our Python or Node.js app that can be easily moved around from system to system with no additional configuration.

After learning about building and running containers, we learned how to manage deploying different kinds of containers because applications rarely exist in a vacuum—web apps often need databases to run. Running and managing more than one container at a time is called container orchestration and is exactly what Docker Compose, Docker Swarm, HashiCorp Nomad, and, of course, Kubernetes do. While each tool handles containers differently, we'll look at a number of ways these tools intersect and when and where you might use them.

Kubernetes is one of the most used container orchestration tools in existence and is, in many ways, better than the alternatives thanks to its third-party ecosystem, as well as managed support by many cloud providers. In some cases, companies offer services to run and manage your containers for you where the underlying technology is actually just Kubernetes.

While deploying to Kubernetes may seem complex, this book aims to break apart this complexity by homing in on the various technologies and deployment techniques available so you can determine which option is best for you and your projects. More often than not, you may come to find that Kubernetes is the best option.

Who should read this book

If you are unsure if Kubernetes is right for you or your project, this is the book for you. Ideally, you have some basic knowledge of writing Python or JavaScript code, so you have the foundation to start with the practical examples in this book.

If you are unsure if containers or Docker containers are right for your applications, this book is also for you. Containers are an essential element of Kubernetes, allowing you to create portable applications that can run on any computer, cloud, on-prem servers, Raspberry Pis, or some mixture of all of these. Containers can help make your apps scale. Kubernetes is one of the tools to help ensure that scale is possible, but it's not the only one. We also learn about Docker Compose, Docker Swarm, and HashiCorp Nomad to run containers.

How this book is organized: a roadmap

The name of this book implies that you are about to embark on a journey, and that was done with purpose. Depending on your background, you can pick up from nearly any part of this journey. While each chapter builds on the previous one, the goal is to allow you to work through this book as your project grows or your curiosity does.

Chapter 1 lays the foundation for what we will do in this book. This chapter is for technical and non-technical readers alike to help understand where modern deployment is and what you can do about it.

Chapter 2 is where we create sample Python and JavaScript via Node.js web applications. These applications serve as stand-ins to nearly any type of application you aim to create. Two different runtimes, Python and Node.js, help us understand the challenges that we face when deploying various kinds of software; these challenges are almost identical but different enough to cause a lot of issues.

Chapter 3 is the first foray into deploying applications. This chapter is all about the manual efforts you will take to deploy code to a server using modern technologies like version control through Git and mature technologies like secure shells and firewalls. The manual nature of this chapter is a rite of passage for many developers because this way is often the scrappiest way to get your application deployed.

Chapter 4 converts a manual deployment into an automated one by leveraging GitHub and GitHub Actions to run various commands on our behalf using a one-off computing workflow. This computing workflow allows us to use the third-party software Ansible to automate how we configure our deployment environments instead of doing them manually.

Chapter 5 is where we start learning about bundling our applications into containers. Containers help us preconfigure environments for our applications so we can more easily move them around with very little additional configuration. While this might feel more complex up front, containerized apps can run wherever there is a container runtime, regardless of whether the application uses Python, Node.js, Ruby, Java, and so on. Container runtimes mean we do not need our deployment systems to have application runtimes installed to work—as in, we can skip installing Python, Node.js, Ruby, Java, etc., in favor of just a container and a container runtime.

Chapter 6 is where we use automation to build and push our containers to a place to store our containers called Docker Hub. From here, we'll learn about our first way of multi-container management, called container orchestration, with a tool called Docker Compose.

Chapter 7 is where we deploy our first containers to a production server, which will require us to configure the server and run our container images through Docker Compose, all orchestrated by GitHub Actions.

Chapter 8 is where we deploy our containers to Kubernetes. In this chapter, we'll learn how to provision a managed Kubernetes cluster across a set of virtual machines. This cluster of machines, coupled with Kubernetes, gives us

the ability to scale and manage containers, unlike any tool we have used to this point. Managed Kubernetes unlocks much-needed production features, such as a static IP address, load balancers, and persistent volumes. We also learn how to design manifests so we can be deliberate about what containers need to do on Kubernetes and how.

Chapter 9 is where we deploy containers to two Kubernetes alternatives called Docker Swarm and HashiCorp Nomad. Both these tools orchestrate containers like Kubernetes, but the approach and features are different. Docker Swarm is a natural extension of Docker Compose, while HashiCorp Nomad is a unique take on managing containers that fits well within the HashiCorp ecosystem of tools like Terraform and Vault.

As you can see, each chapter builds on the concepts introduced in previous chapters and ultimately ends with two chapters covering how to manage containers with modern container orchestration tools. The fundamental building block of Kubernetes is a container and therefore containers are also a fundamental concept for this book. The first few chapters help you understand both conceptually and practically the need for containers, while the remainder of the book helps you better understand and leverage containers in deployment.

About the code

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes code is also **in bold** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (`➞`). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

You can get executable snippets of code from the liveBook (online) version of this book at <https://livebook.manning.com/book/road-to-kubernetes>. The complete code for the examples in the book is available for download from the Manning website at <https://www.manning.com/books/road-to-kubernetes>.

Each chapter, with the exception of chapter 1, uses code and code listings throughout. A few public repositories are available for you at

- <https://github.com/jmitchel3/roadtok8s-py>
- <https://github.com/jmitchel3/roadtok8s-js>
- <https://github.com/jmitchel3/roadtok8s-kube>

The specific chapters will reference these code repositories as needed. The available code is meant to help supplement the book and be used as learning material. The code may or may not be production-ready, and the author cannot guarantee the code will be maintained beyond what is already currently available.

liveBook discussion forum

Purchase of *Road to Kubernetes* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to <https://livebook.manning.com/book/road-to-kubernetes/discussion>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

On my "Coding For Entrepreneurs" blog, I regularly write guides and tutorials related to applications development, containers, infrastructure as code, Kubernetes, and other related software topics. The blog is free and is available at <https://www.codingforentrepreneurs.com/blog/>.

about the author



JUSTIN MITCHEL is a husband, father of three, entrepreneur, coder and founder of CodingforEntrepreneurs.com. He has been teaching web-based software such as Python, JavaScript, Machine Learning, DevOps, and more for the past 10+ years. He has taught over 800k+ students on Udemy, 235k+ on YouTube, 15k+ on GitHub and a semester as an adjunct professor at the University of Southern California.

ABOUT THE TECHNICAL EDITOR

Billy Yuen is a senior architect at Atlassian. He was the lead author for Gitops and Kubernetes book and speaker at Kubecon, Velocity and Java One.

about the cover illustration

The figure on the cover of *Road to Kuberentes*, a “Le Curé Morelos,” or “father Morelos”, is taken from a book by Claudio Linati published in 1828. Linati’s book includes hand-colored lithographs depicting a variety of civil, military, and religious costumes of Mexican society at the time.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

Kubernetes and the path to modern deployment

This chapter covers

- Understanding our path to deployment
- Implementing version control and deployment as documents
- Working with the challenge of dependencies
- Asking what containers do
- Using containers without automated orchestration
- Using Kubernetes with automated container orchestration

Before we can use Kubernetes and its wonderful features, we have to understand the path that leads up to it. This does not mean the history behind the technology but rather the skills behind the fundamentals of deploying software from scratch all the way to Kubernetes.

This book is designed for people with at least a beginner's background in coding modern software applications.

If topics such as functions, classes, variables, and string substitution are new to you, this book might *not* be for you. In that case, please consider taking a beginner object-oriented programming course using either Python or JavaScript. This recommendation is for three reasons:

- Python and JavaScript are incredibly beginner-friendly languages (Python is more so, in my opinion).
- Python and JavaScript are arguably the most widely used programming languages.
- In this book, we'll deploy a Python application and a JavaScript application.

1.1 *Anyone can deploy*

If you can read English, you can deploy software. There are many, many books that make this feel more complex than it needs to be. It is the aim of this book to make deployment possible for you and your team. This book is designed to stack your skills with each chapter so you can take as long as you need prior to moving on.

Once we understand the basics of deployment, we'll start to adopt more and more advanced technologies while maintaining a level of simplicity that, assuming you're following along step-by-step, you will be able to understand.

Kubernetes and Docker are terms that might scare you; they shouldn't. Both are *declarative* tools. Here are examples of declarative statements:

- Use this `insert_your_app_name` Python app.
- Use Python 3.10.
- Run it on Port 80 and 443.
- Allow traffic from the internet.
- Use `example.com`.

These statements are what I want *done* as opposed to *how it should be done*. The *how* is handled by the tool itself. We're concerned with the result, not the process to achieve that result (figure 1.1).

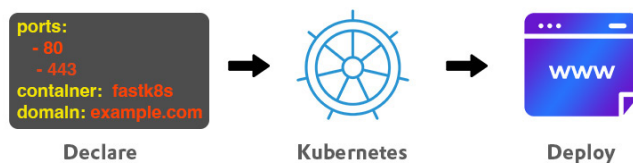


Figure 1.1 Declare to Kubernetes to deploy

Writing the contents of your declarative file(s) can be tricky, but once you get it right, you can use it repeatedly across many of your deployed applications. What's more, you can reuse the files to redeploy to entirely new servers and hosts.

In Kubernetes, these declarative files are called *manifests*. In Docker, they are called *Dockerfiles*. Web Servers like Apache (`httpd`) and NGINX run off declarative files. Popular infrastructure-as-code tools such as Ansible and Terraform are run using declarative files also.

1.2 Our path to deployment

Software is eating the world.

—Marc Andreessen, 2011

Deployed software is the only kind of software that can eat the world; if it's not deployed, how can it bring value to the world?

Software can be deployed on:

- Personal devices, such as smartphones, tablets, desktops, laptops, watches, TVs, cars, refrigerators, thermostats, toasters, RVs, etc.
- The cloud (virtual servers) with providers such as Amazon Web Services (AWS), Linode, Google Cloud Platform (GCP), Microsoft Azure, DigitalOcean, and many others.
- Physical, on-premise servers, such as Linux servers, Windows servers, old desktop computers, Raspberry Pis, Turing Pi, and many others.
- Hybrid deployments on any/all of the above.

There are far too many options to deploy software for this book to cover, so we'll keep it focused on cloud-based and internet-facing software. Regardless of where you deploy, software has the potential to reach *all other* deployed software thanks to the power of networking and the internet.

Our deployment path forward is shown in figure 1.2.

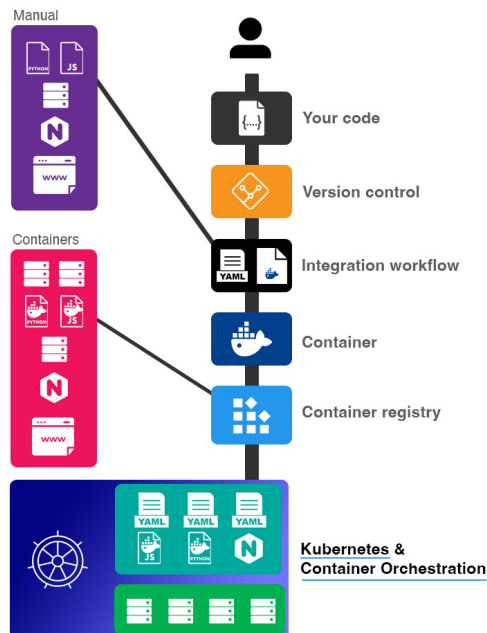


Figure 1.2 Code path to Kubernetes

Throughout this book, we'll learn various deployment methods to fully appreciate the power and simplicity Kubernetes provides.

We start with building two different web applications in Python and JavaScript as tools that we can deploy and continuously iterate on. These applications will be mostly rudimentary, but they will help us tangibly understand each one of the deployment phases.

After our apps are written, we'll use Git, a form of version control, to track and share all changes to our code. With Git, we'll implement a GitHub Repository to store our code, along with a GitHub Actions integration workflow as the basis of full production-ready deployment automation.

Next, we'll adopt containers, commonly known as Docker containers, to help better manage application dependencies and to package, share, and run our code on nearly any machine. We will implement an integration workflow that will automatically build our containers based on our activities in Git. Once built, we'll push (i.e., upload) our containers into a container storage and distribution hub called a container registry.

With our containers stored in a container registry, we'll be able to move to our next phase: container-based deployments. We'll soon learn that these kinds of deployments can be drastically easier than our non-container deployment process, due to how containers manage application dependencies and how effective modern container tools are at pulling (i.e., downloading) and running containerized applications.

Now that we feel like we've unlocked a whole new world of deployment, we quickly realize container-based deployments also have their limits. Deploying more than one container at a time is possible with tools like Docker Compose and Watchtower, but managing upgrades and handling additional traffic (load) becomes unsustainable quickly.

You might think that adding additional compute to handle traffic is a good idea, and often it is, but a fundamental question becomes, are we *efficiently* running our container-based apps? Are we allocating resources well? These questions and many more are solved by adding automated container orchestration to our workflow. One such option: Kubernetes.

Implementing a simple deployment of Kubernetes is, as we'll learn, pretty simple when using a managed Kubernetes Service (as we will). What gets more complex is adding additional services that may, or may not, require internet traffic.

The purpose of this book is to help you understand and solve many of the challenges with the road laid out here.

1.3 *The challenge of dependencies*

As you may know, software comes in many shapes and sizes, whether it's Python, JavaScript, Java, Swift, Objective C, C++, C#, Rust, Ruby, PHP, and many others. Just listing these programming languages can yield many questions:

- Will it run on my device?
- Does it work on a server?

- What version of language X are you using?
- Where do you download or install language X from?
- Is it a beta version or a stable version?
- Did you mean <mature version> or <new forked version>?
- Is this a web application, worker process, native app, or something else?
- What third-party packages are you using?
- Is this open-source software, or do we need to purchase a license?
- What operating system is it going to run on?

These questions all stem from a fundamental concept: *dependencies*.

Regardless of programming language, software developers must wrangle together the various dependencies for their software to work correctly (building, compiling, running, etc.). Dependencies can be

- Operating system (e.g., Xbox One games run on Xbox One)
- Application specific (e.g., FastAPI web apps require Python to run)
- Version specific (e.g., Django version 4.1 requires at least Python version 3.8 and certainly not Python 2.7)
- Third-party dependencies (e.g., The Python package MoviePy requires FFmpeg to be installed on the OS)
- Hardware-specific (e.g., PyTorch deep-learning models require a GPU, the network cable plugged in, the router on, CPUs and RAM, etc.)

This is to say, I cannot simply attach a Blu-ray player to an iPhone and run an Xbox One game without serious modification to one or all of these items. Even if the game itself has a version that is available in the App Store, it has undoubtedly been modified to work on an iPhone specifically. To drive the point home, you might ask, “But what iPhone version?”

It’s no surprise that the previous scenario is done on purpose to aid vendor lock-in to the respective hardware and software platforms. This scenario also highlights how application dependencies and the respective hardware requirements can have serious effect on the success of the software itself.

Kubernetes and much of the software we’ll use in this book is open-source and thus able to be used, modified, and/or changed at will. Full open-source projects do not have vendor lock-in, because that’s kind of the point.

1.4 What do containers do?

Containers package your software and your software’s dependencies into a nice bundle that makes it easier to run, share, and deploy across *most* computing devices. In other words, containers make your software portable. The following listing is a simple example of the configuration file for a container.

Listing 1.1 Dockerfile

```
FROM python:3.10
COPY . /app
WORKDIR /app
RUN python3 -m pip install pip --upgrade && \
    python3 -m pip install --no-cache-dir -r requirements.txt
CMD ["gunicorn", "--bind", "0.0.0.0:8000", "main:app"]
```

In fewer than 10 lines of code, we can define the required isolation for a near production-ready Python application. This is the simple power of containers. This simplicity is true for a *large number* of applications, regardless of their programming language.

The process that follows creating a configuration file is simple as well. We use a *container builder* to build and bundle application code into a *container image* from a configuration file called a *Dockerfile*. Once the image is built, we can either *share* or *run* the *container image* using a *container registry* or a *container runtime*, respectively. *Building*, *sharing*, or *running* containers is often referred to simply as *Docker*, but that doesn't give us the whole story.

Docker is the software and the company behind the concept and technology for containers. Docker and many of the underlying technologies are now open-source, which means Docker is no longer the only way to build, share, and run containers. Here are a few examples of applications that can, and do, run using containers:

- *Programming languages*—Python, Ruby, Node.js, Go, Rust, PHP
- *Web server tech*—NGINX, HTTPD, Kong
- *Databases*—MySQL, PostgreSQL, MongoDB, Cassandra, Redis
- *Message queues*—RabbitMQ, Kafka

Containers have many properties that we'll discuss throughout this book, but the general idea is that if the software is open source, there's a really good chance you can efficiently use containers with that software. If you can use containers, then you can use Kubernetes.

1.5 The challenges in building and running containers without Kubernetes

Starting the build process for a container image is really simple. What's not as simple is figuring out all of *the* dependencies that any given application has: OS-level, application-level, etc. Once you have the dependencies, you can start creating the container configuration file, known as the *Dockerfile*. The complexity that goes into Dockerfiles can be vast and ranges from super simple (as displayed previously) to more intricate (something we'll leave for another book).

The Dockerfile can often give you false confidence that you won't know bugs exist until you *build* and *run* your container image. Even if the *build* succeeds, the *running* can fail. Even if the *running* succeeds initially, it, too, can still fail.

In a perfect world, these failures would be avoided in production. That, of course, is not always possible.

To make matters even trickier, updating an application's code to an already-running container image is rarely graceful and can require downtime without the aid of other tools. This is yet another problem we want to avoid in production as much as possible.

Here's a few highlights of challenges that we will likely face when using just containers without Kubernetes:

- Updating or upgrading container versions
- Managing three or more containers at once
- Scaling container-based apps and workloads
- Failure recovery
- Container-to-container communication (interoperability)

All of these challenges and many more are often solved by using Kubernetes, and these are topics we explore throughout this book.

1.6 *What are the fundamental challenges with using Kubernetes?*

At this point, you might believe that Kubernetes is the end-all, be-all solution for deployment. It isn't. The major challenges in Kubernetes adoption boil down to

- Installing and updating
- Configuration overload
- Stateful vs. stateless applications
- Security and permissions
- Container-to-container communication
- Storage
- Recovery and debugging

All this friction might cause developers to look elsewhere to deploy applications.

Kubernetes typically runs containers across a cluster, or group, of virtual machines. By default, Kubernetes will decide for us which virtual machine, or node, our containers will run on. Installing and updating the Kubernetes software itself is one of the many reasons that we're going to opt for managed Kubernetes in this book.

Once we have Kubernetes running, we are now faced with a non-trivial amount of configuration, even to run just one container-based web application. To update even one application, we'll have to learn how to implement continuous integration and continuous delivery (CI/CD) tools and role-based access control (RBAC) to make changes to our Kubernetes cluster.

Containers are stateless by design, which means stateful applications (like databases) are another layer of complexity developers might not be ready for. Many stateful

applications were not designed for container-based or even cluster-based operations, so using them poses another set of unique challenges.

Cross-container communication, recovery, and debugging are other layers of complexity that we'll explore in this book.

1.7 **How does this book teach the technology?**

This book will teach the technology with step-by-step code examples and guides to practically deploy two production-grade web applications written in Python and JavaScript, respectively, using the following approaches:

- Manual deployment
- Container deployment
- Kubernetes Deployment

Each type of deployment stacks skills and tools from the previous deployment.

The benefit of Kubernetes comes full circle once you attempt to manually deploy applications continuously to virtual machines. Manual deployment is important because it gives you the exact context of what is behind automated deployments. Automating something you don't understand will leave you vulnerable when mistakes or bugs inevitably happen because *everything goes wrong all the time*.

The reason we use Python and JavaScript is merely for their popularity among developers (and the author). If you're interested in another programming language, please consider our web application demos on this book's related GitHub (<https://github.com/roadtokubernetes>). Remember, once your application is containerized, you can almost certainly use two of these three deployment options.

The second half of this book will be dedicated to discussing some of the intricacies of running and maintaining a Kubernetes Deployment so we can continuously integrate new features and applications into our Kubernetes Deployment.

Summary

- Deployment can be complex, but anyone can do it.
- For continuous updates, we need an automated pipeline for deployment.
- Python and JavaScript apps are among the easiest web applications to deploy.
- App dependencies may or may not be obvious when first deploying.
- Containers help isolate our apps for sharing and running.
- More servers, not more robust ones, are often the solution for handling traffic increases.
- Adding more containerized applications becomes increasingly difficult.
- Kubernetes reduces complexity by managing the deployment of containers while distributing compute resources effectively.

Creating the Python and JavaScript web apps

This chapter covers

- Designing a basic FastAPI web app in Python
- Isolating Python projects with virtual environments
- Designing a basic Express.js web app in JavaScript and Node.js
- Initializing new Node.js projects using npm
- Tracking code changes with Git
- Pushing code to GitHub

The goal of this book is to understand how to deploy software ultimately using Kubernetes. This chapter sets the stage by giving us two different applications to deploy with. These applications will be continuously referenced throughout this book and are meant to be a stand-in for the actual applications you intend to deploy.

We'll use the JavaScript runtime Node.js and the Python programming language as the foundation for these applications. These two languages are incredibly popular and easy to work with. They also have a number of overlapping features, but not identical ones, which require us to adjust how we think about the various nuances that go into deploying each of them. For as much as I enjoy working with these languages, this book is by no means meant to be the end-all-be-all for building and deploying applications in either Node.js or Python.

We will soon learn that deploying applications, regardless of their programming language or runtime, can be a complex task regardless of the complexity of the application itself. Over the course of this book, we'll learn that containers help simplify this complexity by providing a consistent way to package and deploy applications. Before we can learn about containers, we must learn how to build basic applications.

2.1 *Designing a basic FastAPI web app in Python*

FastAPI is a web application framework that makes it simple to build HTML-driven or API-driven websites with Python. We use web frameworks like FastAPI, so we do not have to rethink or rewrite many of the common features across what websites need to have. FastAPI is simply a third-party Python package that has already implemented many of these features, so we can focus on writing a few Python decorators and functions to power a fully functional web application. We'll approach building our Python-based website app with the following tools:

- *Python version 3.8 or higher*—If you need help installing Python, please review appendix A.
- *venv*—A Python module for creating and managing virtual environments for isolating code.
- *FastAPI*—A popular Python web framework that comes with a minimal amount of features to build a highly functional application.
- *uvicorn*—The web server we will use to handle web traffic with FastAPI.

Each one of these tools is set up and installed in the order presented. To continue, I will assume that you already have Python installed on your system and you are ready to create a virtual environment for your Python project. If you are experienced with Python, you might want to use different tooling than what I listed, which is exactly what I want you to do. If you are new to Python, I recommend you follow along with my configuration until you learn more about Python.

2.1.1 *Python project setup with virtual environments*

Like many programming languages, Python has a rich developer ecosystem with many different third-party packages and libraries and, as a result, many different versions of them. To help keep track of these versions and to help isolate our Python projects from one another, we use a concept called virtual environments. Virtual environments are not true isolation, but they do help keep our projects organized and help us avoid version conflicts.

In this book, our Python applications will use the built-in virtual environment manager known as *venv*. Let's set up a folder for our Python project and create a virtual environment.

Open your system's command line interface (CLI) and navigate to a folder where you want to store your Python project. For this book, I'll be using the `~/Dev/roadtok8s/py/` folder (listing 2.1). If you're on macOS or Linux, you will open Terminal.

If you're on Windows, you can use PowerShell or the Windows Sub-system for Linux (WSL).

Listing 2.1 Creating a Python project development folder

```
# Make the Python project folder
mkdir -p ~/Dev/roadtok8s/py/

# Create a folder for our Python source code
mkdir -p ~/Dev/roadtok8s/py/src

# Navigate to our Python project folder
cd ~/Dev/roadtok8s/py/
```

Every time you create a new Python project, I recommend you follow these same steps to help isolate the code for different projects from one another. To help further isolate your code, let's create a virtual environment for our Python project.

Naturally, at this point I assume you know which Python interpreter you will use. If you're on macOS or Linux, you will likely use `python3` instead of `python` and if you're on Windows you will likely use `python` instead of `python3` (listing 2.2). You can learn more about this in appendix A for Mac users or appendix B for Windows users.

Listing 2.2 Creating a Python virtual environment

```
# In our Python project folder
cd ~/Dev/roadtok8s/py/

# Windows users
python -m venv venv

# Linux or macOS users
python3 -m venv venv
```

Once this command is complete, you'll see a new directory called `venv`, which will hold all of the Python packages and libraries we will install for this project. To use this new virtual environment and all the related tools, we will need to activate it. You will activate the virtual environment every time you want to use it, and some text editors (such as VSCode) may activate it for you. Let's activate our virtual environment now, as shown in the following listing.

Listing 2.3 Activating a Python virtual environment

```
# navigate to our Python project folder
cd ~/Dev/roadtok8s/py/

# windows users
.\venv\Scripts\activate

# linux or mac users
source venv/bin/activate
```

Now that it's activated, we can start installing our project's third-party dependencies using the Python Package Manager (pip). Be aware that pip might be installed globally on your system, but we need to use our virtual environments version of pip. To ensure this is the case, we use the virtual environment's python interpreter to run pip. The following listing shows how it's done.

Listing 2.4 Installing Python packages

```
# With the virtual environment activated
$(venv) python --version

# Update pip
python -m pip install --upgrade pip

# Install FastAPI
python -m pip install fastapi
```

← In this example, `$(venv)` signifies that the virtual environment is activated. Your command line may appear differently.

Installing each package one at a time is fine for a small project, but as your project grows, you will need to install many more packages. For this, we'll use another feature of pip that allows us to reference a file full of packages to install; this file is called a *requirements* file and is most often labeled as *requirements.txt*. In your Python project's source code folder (e.g., `~/Dev/roadtok8s/py/src/`), add the following to a file called *requirements.txt*, as shown in the following listing.

Listing 2.5 FastAPI requirements.txt

```
fastapi
jinja2
uvicorn
gunicorn
```

With this file, we can use the `-f` flag built into pip to install all the packages listed in the file. This requirements file can also declare specific versions of code, but we'll leave that for another time. See the following listing to install third-party packages.

Listing 2.6 Installing all third-party packages

```
# navigate to root of our project where the venv folder lives
cd ~/Dev/roadtok8s/py/

# activate or reactivate your virtual environment
source venv/bin/activate # or .\venv\Scripts\activate

# install all packages via the newly created requirements file
python -m pip install -r src/requirements.txt
```

While this requirements file might seem trivial, it's important to have one so we can attempt to recreate the necessary conditions to run our code on other machines. Requirements files can become a lot more complex and include things like version

numbers, version ranges, or even Git repositories. Complex requirements files are outside the scope of this book, but if you're interested, I recommend reviewing the open-source Python package called pip-tools (<https://github.com/jazzband/pip-tools>).

Now that we have established the requirements in our environment, we have two options: (1) Destroy the virtual environment and try again so you learn how easy it is or (2) start writing our FastAPI code. Depending on your experience, I'll let you decide.

2.1.2 Hello World with FastAPI

I believe that FastAPI became popular within the Python community because it took an API-first approach coupled with advanced asynchronous programming while remaining familiar and simple to use. Both features are great for developers who want to create a REST API that can handle a lot of requests and responses.

To create our FastAPI application, it's a matter of just a few imports, initializing a class, and defining functions that map to HTTP URL routes and HTTP Methods. To do this, we'll create a Python module named *main.py* in our Python project's *src* folder we created earlier. The following listing shows a look at the code.

Listing 2.7 Creating *src/main.py*

```
# Import the FastAPI class from the fastapi module.
from fastapi import FastAPI

# Declare an instance of the FastAPI class.
app = FastAPI()

# use the app instance as a decorator to handle an
# HTTP route and HTTP method.
@app.get("/")
def read_index():
    """
    Return a Python Dictionary that supports JSON serialization.
    """
    return {"Hello": "World"}
```

As we see how simple this code is to write, it has a key distinction that many other web frameworks don't: the function returns a dictionary value. Most web frameworks are not this simple by default. This dictionary value is converted into JSON data, which is a data format that is often used in REST API software development. The Python dictionary itself needs to be JSON serializable, which is typically straightforward but can complicate things from time to time. To test dictionary serialization with Python, it's as simple as `python -c 'import json; json.dumps({"your": "dictionary_values"})'`.

Now that we have our first Python module, let's run it as a web server with the aid of *uvicorn*. *uvicorn* is a lightweight gateway server interface that makes it possible to handle asynchronous requests and responses. In later chapters, we will use *uvicorn* coupled with the production-ready Web Server Gateway Interface (WSGI) *gunicorn* to create a highly performant web server for our FastAPI application. Here's a standard format in

the following listing for using uvicorn to run FastAPI applications during the development phase.

Listing 2.8 uvicorn syntax

```
uvicorn <module>:<variable> --reload --port <port>
```

To adjust this to our project, the `<module>` corresponds to `src.main` because the `main.py` file is located in `src/main.py`. The `<variable>` will correspond to the instance of `FastAPI()`, which happens to be `app` in `src/main.py`. The `--reload` flag will restart the application when we save any files related to it. The `--port` flag will allow us to specify a port number for our application to run on. Let's see a practical example of this syntax in action in the following listing.

Listing 2.9 Starting FastAPI with uvicorn

```
# navigate to python project
cd ~/Dev/roadtok8s/py/

# activate or reactivate your virtual environment
source venv/bin/activate # or .\venv\Scripts\activate

# run the FastAPI application with Uvicorn
uvicorn src.main:app --reload --port 8011
```

The `PORT` (e.g. `--port 8011`) you use for any given application becomes increasingly important as you start running more projects. With uvicorn running our FastAPI application at port 8011, we can open our web browser to `http://127.0.0.1:8011` and review the results (figure 2.1).



Figure 2.1 Uvicorn running FastAPI at port 8011

For some, getting to this point is a huge accomplishment. For others, it's a bit mundane. Wherever you fall on the spectrum, I challenge you to pause for a moment and add a few new HTTP URL routes to your FastAPI application before moving on.

2.1.3 Handling multiple routes with FastAPI

Most web applications have many more routes than the one we just created. With Python and FastAPI, it's incredibly easy to add new routes and HTTP method handlers. Let's add a new route to our FastAPI application that responds to the `GET` method at the `/api/v1/hello-world/` route, as shown in the following listing.

Listing 2.10 Adding a new route in src/main.py

```

@app.get("/api/v1/hello-world/") ← Defines a new route for the GET HTTP method
def read_hello_world(): ← Shows the unique function name
    """
    Return an API-like response.
    """
    return {"what": "road", "where": "kubernetes", "version": "v1"} ← Returns a Python dictionary value
                                                                    that will be converted to JSON

```

Assuming you still have uvicorn running, open your web browser and visit `http://127.0.0.1:8011/api/v1/hello-world/` and you should see something similar to figure 2.2.



Figure 2.2 Uvicorn running FastAPI at port 8011

Unsurprisingly, this response is identical to listing 2.7 with different data, and that's exactly the point. Creating new routes and HTTP method handlers is this easy with FastAPI.

When it comes to building web applications, a big part of how you design them will fall in the URL route (e.g., `/api/v1/hello-world/`) and the HTTP methods (e.g., `GET`, `POST`, `PUT`, `DELETE`, etc.). FastAPI makes it easy to handle both features.

Now that we have a basic understanding of how to create a FastAPI application, let's move on to creating a Node.js application.

2.2 Creating a JavaScript web application with Node.js and Express.js

Express.js is a web framework for Node.js that makes it easy to create web applications. Express.js will feel almost identical to FastAPI, except instead of using Python, we'll use JavaScript. But what kind of JavaScript are we talking about? There are two types of JavaScript: browser-based JavaScript and server-side JavaScript.

Browser-based JavaScript is the JavaScript that runs within your browser. It's how we can build dynamic user interfaces, and it's the reason we have tools like React.js, Vue.js, and Angular. Node.js and, by extension, Express.js, are not browser-based JavaScript tools.

Server-side JavaScript is known as Node.js and runs on the server. Much like Python is a runtime for the Python programming language, Node.js is a runtime for the JavaScript programming language.

Express.js, like FastAPI, is a server-side web framework that helps us build web applications with minimal overhead. To use Express.js, we will need to have the following installed:

- *Node.js version 16.14 or higher*—If you need a reference to install Node.js, please review appendix B.
- *The Node Package `npm`*—This is a built-in Node.js package for installing and managing third-party packages (much like Python’s `pip` package).

Assuming you have Node.js on your machine, let’s create a folder in the following listing for our JavaScript project and install Express.js.

Listing 2.11 Creating a folder for this chapter’s JavaScript code

```
mkdir -p ~/Dev/roadtok8s/js/src/  
cd ~/Dev/roadtok8s/js/
```

The Node Package Manager (`npm`) defaults to isolating your code to the local directory. In other words, you do not need to *activate* anything to install packages in your local directory. What’s more, `npm` will automatically keep track of the packages you install in a file called `package.json`.

To start a new Node.js project, it’s typical to run `npm init` in the root folder of your project because it will help with the beginning setup of your project. Fill out the questions it prompts you with as you see fit. Get started in the following listing.

Listing 2.12 Initializing a new Node.js project

```
npm init
```

After you complete this process, you’ll see the output shown in the following listing.

Listing 2.13 Output from `npm init`

About to write to `/Users/justin/Dev/roadtok8s/js/package.json`:

```
{  
  "name": "js",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC"  
}
```

In this case, I typically swap `"name": "js"` for `"name": "roadtok8s"` because `js` is far too generic a name.

You can also see this data in `package.json` by running:

```
cat package.json
```

From here on out, when you run `npm install <package-name>`, npm will automatically add the package to `package.json` for you, thus saving you the trouble of manually adding it. Let's install Express.js now in the following listing.

Listing 2.14 Installing Express.js

```
npm install express
```

Once this completes, we can review `package.json` and see that npm has added Express.js to our project, as shown in the following listing.

Listing 2.15 Output from package.json

```
...
  "dependencies": {
    "express": "^4.18.2"
  }
...

```

← Your version of Express.js may be different

As we see, npm has a standardized way to both *install* and *track* package dependencies. This is a bit more rigid than Python's pip but certainly cuts down the complexity of managing development environments yourself and/or depending on another third-party tool to do it. Now, verify the contents of `~/Dev/roadtok8s/js/`, as shown in the following listing.

Listing 2.16 List the contents of the node-web-app directory

```
ls ~/Dev/roadtok8s/js/
```

At minimum, the following should be listed in our directory:

- `node_modules/`—Contains all of our third-party packages. Treat this as a folder you can delete anytime and reinstall with npm install.
- `package-lock.json`—This is an auto-generated file from `package.json` with more detailed information about our project.
- `package.json`—Contains the versions of all of the third-party packages installed, our package (or project) name, our package's version, and any custom scripts we may add.

Now that our environment is set up for our Node.js project, let's create our first module for our web app.

2.2.1 Hello World from Express.js

Much like with FastAPI, we'll start with a simple Hello World application using Express.js. To do this, we'll need to create a new file called `main.js` in our source code (`src`) directory of our JavaScript project folder (`~/dev/roadtok8s/js/`).

Unlike FastAPI, Express.js will not use a third-party web server like uvicorn, so we'll use a built-in feature of Express.js to listen to a specific port on our machine. The process goes like this:

- 1 Import the necessary modules for Express.js, the file system, and path.
- 2 Create an instance of the Express.js module.
- 3 Define a default port for the web app to listen to or tie it to an environment variable for PORT.
- 4 Create a callback function to handle an HTTP URL route and specific HTTP Method.
- 5 Define an App start-up function to listen to a port, output the process ID to a local file, and output a message to the console.

The following listing shows what that code looks like.

Listing 2.17 Express.js module in main.js

```
const express = require('express'); // ← Use require() and not
const fs = require('fs');           import. This is a Node.js
const path = require('path')         thing.
const app = express();
const port = process.env.PORT || 3000; // ← A default PORT value is
                                       required to run this app.

app.get('/', (req, res) => {
  res.send("<h1>Hello Express World!</h1>");
})

app.listen(port, () => {
  const appId = path.resolve(__dirname, 'app.pid')
  fs.writeFileSync(appId, `${process.pid}`);
  console.log(`Server running on port http://127.0.0.1:${port}`);
})
```

The format Express.js uses to handle any given HTTP URL route, HTTP Method, and callback function is as shown in the following listing.

Listing 2.18 Express.js HTTP handling

```
app.<httpMethod>(<pathString>, <callback>)
```

- **<httpMethod>**—This is the HTTP method that will be handled by the corresponding callback function. In main.js, this is simply the GET method.
- **<pathString>**—This is a string that represents the URL path that will be handled by the corresponding callback function. In listing 2.17, we only handle the index route (/).
- **<callback>**—This is another function that will be called when an HTTP request is made to the corresponding URL path (e.g., /).

Now that we have our `main.js` module configured and ready to accept a URL request as well as run on a specific port, let's run it and see what happens.

2.2.2 Run the Node.js + Express.js app

To run a JavaScript on the command line in the backend is as simple as `node <module>`. At this time, we'll navigate to our Express.js source code and run `node main.js` to start our application. By default, our Express.js app will listen to port 3000.

Listing 2.19 Running the Express.js web server

```
cd ~/Dev/roadtok8s/js/src
node main.js
```

This command should yield `Server running on port http://127.0.0.1:3000` in the terminal. Let's open that URL in our web browser and have a look (figure 2.3).



Figure 2.3 Express.js running at `http://127.0.0.1:3000`

To look at the HTML source code, on a web browser, look for `view source` for the page in developer tools to see the output, as in figure 2.4.

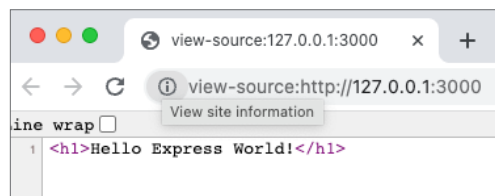


Figure 2.4 HTML source code of `http://127.0.0.1:3000`

The source code shows us two things about our Express.js application:

- Express.js can easily return HTML.
- Incorrectly formatted HTML will still be rendered by the browser.

The PORT we run our apps on becomes increasingly important when we start deploying applications. With that in mind, let's change the port our Express.js app runs on.

Press Control + C and add the PORT environment variable of 3011 to test a new port, as shown in the following listing.

Listing 2.20 Running the Express.js web server on a new port

```
# macOS or Linux
PORT=3011 node main.js

# Windows
set PORT=3011 && node main.js
```

The console output should now reflect the new port: `Server running on port http://127.0.0.1:3011`. Open your browser to this new URL and see what happens (figure 2.5).

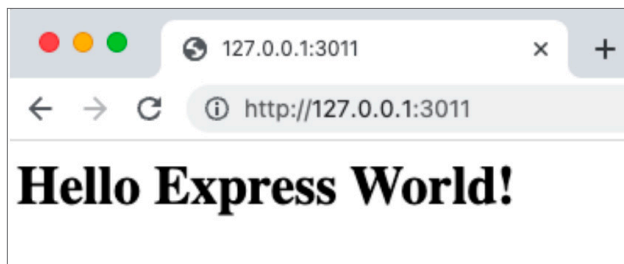


Figure 2.5 Express.js running at `http://127.0.0.1:3011`

What if you tried to visit `http://127.0.0.1:3000` again? Does anything happen? This is the early stages of seeing how much of an affect the PORT that you choose can have on our applications and our ability to access it.

Options to Stop/Cancel the running Node.js server:

- Exit the command line / terminal window (easiest).
- Use Control+C in the terminal where the program is running (recommended).
- Use `kill -9 <PID>` where <PID> is the process ID of the running Node.js instance (advanced).

Now that we can render HTML responses with Express.js, let's create a JSON response by using built-in JavaScript features.

2.2.3 **Create a JSON response with Express.js**

Express.js is often used as a microservice, which means that it's a great way for software to communicate with software, and that's exactly what JSON data is for. There are two ways to return JSON data with Express.js:

- Using `res.json()`
- Using headers

We'll leave using headers for another time and opt for the much simpler version of `res.json()`. To return JSON data within Express.js, we need to do the following:

- Declare a JavaScript object (or retrieve one from a database).
- Convert the object into JSON using `JSON.stringify()`. Oddly enough, JavaScript objects are not the same as JSON strings.
- Return the JSON string using `res.json()`.

Here's the resulting callback function for our new route in the following listing.

Listing 2.21 Adding a route in `main.js`

```
app.get('/api/v2/rocket-man1/', (req, res) => {  
  const myObject = {who: "rocket man", where: "mars"};  
  const jsonString = JSON.stringify(myObject);  
  res.json(jsonString);  
})
```

With node `main.js` still running, navigate to our new route, and you should see the same output from figure 2.6.



Figure 2.6 JSON response from Node.js

You should now have a working Express.js application that can handle both JSON and HTML responses, which is a great start for our applications. As with many things, we could spend book-lengths on Express.js, but we'll leave it here for now.

With both of our applications, we must start tracking our code changes and pushing them to a remote repository. The sooner you get in the habit of doing this, the better.

2.3 Tracking code changes with Git

Git is an open-source tool that has a number of features that make updating, sharing, and tracking code far easier. Git is a type of version control system that is often referred to as source control because it tracks the source code of many applications.

Git is certainly the most popular version control system in the world and is really a fundamental tool that powers open-source technology. I used Git to help write the many saved iterations of this book. The primary features we will focus on with Git in this book are

- Track changes to our code
- Ignore certain file types
- Revert to previous versions of our code
- Push (upload) our code to a non-local storage location (called a remote repository)
- Pull (download) our code when we go into production
- Trigger workflows to use our code (such as CI/CD and other automation)

These features will be used at various times throughout the book to continuously aid us with deployment in general and our journey to Kubernetes.

Be sure to install Git on your local machine before continuing. Visit <https://git-scm.com/downloads> because it's pretty straightforward to install and get started. To verify that Git is installed, run the following listing's command in your command line.

Listing 2.22 Verifying Git is installed

```
git --version
```

This should respond with a version number (e.g., `git version 2.30.1`) and not an error. If you get an error, you'll need to install Git as previously mentioned.

What is version control?

Writing in a notebook and making copies for a filing cabinet is an old-school way of doing version control.

Let's take a modern yet simple example: Control + Z (or Command + Z on macOS). If you're anything like me, you love the *undo* keyboard shortcut and probably use it often. This is a form of version control as it's reverting your text to a historical state. *Redo*, Control + Shift + Z or Command + Shift + Z (macOS), is also a form of version control as it's moving forward in time to a more recent state of your text.

How difficult would life be without *undo* and *redo*? I can't even imagine. If you grew up in the late 1900s, you might remember how difficult it was before *undo* and *redo* existed, because maybe you used an actual typewriter with white-out on your documents.

Version control for documents takes this concept to a whole new level and really unlocks the potential of how individuals and teams can effectively write code.

Version control is much like the *undo* and *redo* keyboard shortcuts, but it's for entire files and folders. There's a catch: to use version control, you need a few things:

- A tool installed that is made for version control
- To write the correct commands to use the tool
- To remember when and how to use the tool

Over the years, there have been many tools to help with version control. The tool we'll use is also the most popular: Git.

To manage our Git-based projects, we use what's called a remote repository. In this book, we'll use two types of remote repositories: self-managed and third-party managed. We'll implement a self-managed repository in the next chapter to aid in manual deployment.

GitHub.com is a third-party managed remote repository tool that allows you to store your code in the cloud, share your code with others, and trigger various automation pipelines that we'll discuss more throughout this book. GitLab is another popular remote repository that can be third-party managed or self-managed but is outside the scope of this book. Now, let's update our projects to start using Git.

2.3.1 Using Git in our projects

Getting started using Git is a straightforward process. You start by telling Git to start tracking changes in each of the project directories we created in this chapter: `~/Dev/roadtok8s/py/` and `~/Dev/roadtok8s/js/`. This process is referred to as initializing a Git repository. Let's start with our Python app in the following listing.

Listing 2.23 Using `git init` to initialize a Git repository

```
cd ~/Dev/roadtok8s/py/  
git init
```

Our Node.js app is shown in the following listing.

Listing 2.24 Using `git init` for our Node.js app

```
cd ~/Dev/roadtok8s/js/  
git init
```

Both of these will respond with something like `Initialized empty Git repository in . . .`, naming the path to your repo.

Easy enough right? We see that `git init` command creates what's called a *repository* (*repo* for short) within the directory you run `git init` inside of.

We now have two repos in our `~/Dev/roadtok8s/` directory: `py` and `js`. These repos are currently only stored on our local machine. Soon, we'll configure them to be stored in a remote repository.

If you were to run `git init` in a directory that already has a repo, you'll get a message such as `Reinitialized existing Git repository in . . .`. I have seen this output a great many times. I am sure you will, too.

If you do not know if your directory has a repo, you can simply run `git status`. If you see `fatal: not a git repository`, then you have your answer. If you see nearly anything else, there's a good chance the folder is already in a Git repo. Let's expand on `git status` a bit to better understand what it does.

FILE AND GIT STATUS

Git is a passive tool that requires user input, which means you need to tell it *exactly when* something has occurred and *what* you want done to that change.

So far, we only told Git to *exist* in our project folders (as a repo). Before we can start telling Git to track files, we'll use the command `git status` within our Python project directory. `git status` will yield what we see in figure 2.7.

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    src/
    venv/

nothing added to commit but untracked files present (use "git add" to track)
```

Figure 2.7 Git status for our Python project

Figure 2.7 shows us a few things:

- We are working on a Git branch called `main` or `master`, depending on your configuration. Branches are a way to manage different versions of your code. Branching in Git is beyond the scope of this book.
- No commits yet means we have not told Git to track any changes to our code yet.
- Untracked files provide a list of folders (that include files) along with files that we could either track or ignore (by adding them to a `.gitignore` file)

If we repeat this process for our Node.js app, we should see roughly the same result from `git status` as seen in figure 2.8.

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    node_modules/
    package-lock.json
    package.json
    src/

nothing added to commit but untracked files present (use "git add" to track)
```

Figure 2.8. Git status for our Node.js project

Some text editors (like VSCode) make using Git much easier with the built-in Git features. If, at any time, Git starts to feel too challenging for your current skill level, I highly recommend downloading VSCode and reviewing how to use Git with VSCode.

In both of our repos, we have a list of *untracked files*. Before we start tracking any of these files, we actually want to ignore the vast majority of them, thanks to the third-party packages stored in `venv/` and `node_modules/`.

2.3.2 Ignoring files

There are two *massive* folders in each of our repos (project directories) that we *never want to track* with Git. These folders are

- Python's `venv/`
- Node.js's `node_modules/`

These directories can be *huge* because they are third-party packages that may or may not have a lot of weight (file saves, number of files, etc.) as well as a number of their own dependencies.

Both of our projects have references to the third-party packages that have been stored in `venv/` and `node_modules/`. Those references are `requirements.txt` and `package.json`, respectively. These folders can be *deleted* and *recovered* at any time simply by running `python -m pip install -r requirements.txt` and `npm install`, respectively.

We hope that the third-party packages we're using also have Git repositories that are accessible by the world at large; thus, grabbing copies of them is simple enough. In other words, we'll let the third-party packages be responsible for their own code. Yes, there are caveats to this ethos, but those are outside the context of this book.

To ignore these folders and any other files, we need to create a `.gitignore` file in the root of our repo (next to where we ran `git init`). Notice that the filename starts with a period (`.`) and is often referred to as dot-git-ignore.

First, let's review what is in our Python web app folder/directory:

```
cd ~/Dev/roadtok8s/py/
ls -aa
```

Windows users can use
`dir /a` instead of `ls -aa`.

This should output

```
.git    src    venv
```

If you don't see `.git`, you did something wrong during the `git init` phase. The `.git` directory is where all of the Git tracking information is stored.

If we run `git status` at this time, we'll see that `venv` is listed in the *untracked files*. We'll create a `.gitignore` file specifically for our Python project to prevent `venv` from being tracked. Ensure that `.gitignore` is in the same folder/directory as the `.git` directory, as shown in the following listing.

Listing 2.25 Python gitignore example

```
venv/  
.env  
.venv  
  
__pycache__/  
*.py[cod]  
*$py.class  
  
.DS_Store
```

This is a simple example of a Python `.gitignore`. I recommend using the official Python `.gitignore` file on GitHub (<https://github.com/github/gitignore/blob/main/Python.gitignore>).

Run `git status` and verify that the directory `venv` is no longer there. If you still see `venv` that means that `.gitignore` is either not saved or in the incorrect location.

Let's create another simple `.gitignore` file for our local project in the following listing. As with Python, I recommend using the official node `.gitignore` on GitHub (<https://github.com/github/gitignore/blob/main/Node.gitignore>).

Listing 2.26 Node.js gitignore example

```
node_modules/  
.env  
.venv  
  
logs  
*.log  
npm-debug.log*  
  
.DS_Store
```

Once again, run `git status` and verify the `node_modules` folder is no longer listed. If you still see `node_modules` that means that `.gitignore` is either not saved or in the incorrect location. Ignoring files is incredibly important so we don't accidentally make our remote repository too large *or* accidentally upload sensitive information to a remote repository.

2.3.3 Tracking files

Tracking files is a mostly manual process and should be done frequently to help us understand the changes we have made over time. This tracking process is easy to lose sight of as a beginner because you might be thinking to yourself, "I've already saved the file. Why should I also track it?"

Saving the file and tracking the changes are two very different things. Tracking will denote what was added or removed from a file (or directory) at various points in time based on when you decide to track the changes with Git. To get a better sense of this

process, let's start tracking our Python project's files. First, we'll navigate to our local repo, and then we'll add each file we want to track with the command `git add <path>`.

`git add` does a few things:

- Prepares Git for the files/folders we want to track right now.
- Takes the files or files within the folders at their current save state (like what's actually in them when you run `git add my/file/path.txt`).
- Allows you to "unprepare" them with `git reset <path>` like `git reset src/main.py`.
- Does not actually track, save, or solidify the changes yet. This comes later.

The following listing shows an example of tracking our Python project's files.

Listing 2.27 Using `git add` to track Python files

```
cd ~/Dev/roadtok8s/py/  
git add src/main.py  
git add src/requirements.txt
```

If we think of `git add <path>` as `git prepare <path>`, then we're on the right track. The reason for this step is to ensure we are tracking the files we want to track and not tracking files we don't want to track.

I think of Git as a way to track incremental changes that are interconnected in some way. `git add` is a simple way to ensure we only bring the files and folders that are somehow interconnected. In other words, if I add two features that are unrelated, I should consider adding them and tracking their changes separately. This is a good practice to get into because it will help you and others understand the changes you've made over time.

To finalize the tracking process, we need to *commit* the changes we've made. Committing is a bit like declaring, "Oh yes, those files we want to track, let's track them now". The command is `git commit -m "my message about these changes"`.

`git commit` does a few things:

- Finalizes, solidifies, or saves changes since the last commit based on what was added with `git add`
- Takes a snapshot of the files or files within the folders we added with `git add`
- Requires a message to describe what happened to these added files since the last commit

Here's a working example of committing our Python project's files:

```
git commit -m "Initial python project commit"
```

From here on out, every time you make incremental changes to your app, you'll want to add and commit those changes to your repo. You can remember the process like this:

- 1 Make changes to files or folders in our project.
- 2 Ask, “Are there files that *do not* need to be tracked?” If so, update `.gitignore`.
- 3 Prepare Git for the changes with `git add`.
- 4 Solidify the changes with along with a message about the changes `git commit -m "my message about these changes"`.
- 5 In a future chapter, we’ll run the process of uploading our code elsewhere with `git push`.

I purposefully left out a few features of Git (such as `git stash` or `git checkout`) primarily to focus on the fundamentals of tracking files. I recommend reading the official Git documentation to learn more about these features: <https://git-scm.com/docs>. Let’s put our skills to use and do the same for our Node.js project’s files in the following listing.

Listing 2.28 Using `git add` to track Node.js files

```
cd ~/Dev/roadtok8s/js/  
git add src/  
git add package.json  
git add package-lock.json
```

As we see here, `git add src` shows us how to track an entire directory, which will add all files and sub-folders and the sub-folders’ files within `src/` unless it’s in `.gitignore`. We can solidify our changes by committing them with the command `git commit -m "Initial node.js project commit"`.

Great work! We now have our files tracked by Git. If we *ever* make changes in the future the process is as simple as

- 1 Adding the file we want to track with `git add relative/path/to/file`.
- 2 Committing our changes with `git commit -m "Commit message"`.

Or, we can do so by

- 1 Adding the folder we want to track with `git add relative/path/to/folder/`.
- 2 Committing our changes with `git commit -m "Commit message"`.

You can add and commit files as often as you want or as infrequently as you want. As a general rule, I recommend committing as often as possible with a commit message that describes the changes you made.

For example, let’s say our Python project needs a new URL route. We can create it in our Python module and then let Git know about the new route with a message like “Added new URL route to render HTML content.”

Another good example would be if we wanted to handle a new HTTP method for ingesting data on Node.js. We could create that new URL in our JavaScript module and commit it with a message like “support for ingesting contact form data via a POST request”.

A bad example would be if we added a bunch of features to our project in a bunch of different files and just added a message with “stuff changed”.

These examples are not meant to be prescriptive, but rather to illustrate the importance of writing good committing practices coupled with verbose commit messages.

2.3.4 Complete Git repo reboot

Time and time again I see beginners give up on using Git correctly simply because they made a few mistakes with their repos. When this happens, I recommend rebooting the entire repo. This is *not good practice* in general, but it’s a great practice for those of us still learning Git and many other deployment tools.

What I am suggesting is that we reboot our Git repos by removing all traces of our old Git repos and starting from scratch. Doing this creates undue problems that can be avoided by learning more about Git, but it is a great way to continue learning many of the other technologies in this book. Git is certainly a tool I recommend getting better at, and if you use the tools in this book, you likely will. Git should not be the reason your learning progress stops on how to deploy software.

Let’s see an example of how we can reboot our Git repo for our Python project. this process is the same for our Node.js project:

- 1 Navigate to our Python project directory (`cd ~/Dev/roadtok8s/py/`).
- 2 Destroy the `.git` folder:
 - *macOS/Linux*: `rm -rf .git`. The command `rm -rf` will delete entire folders and all the files within them.
 - *Windows*: `rmdir /S .git`. The command `rmdir /S` (capital S) will delete entire folders and all the files within them.
 - *Note*: Do not use these commands lightly because they can blow up in your face if you do them in the wrong folder.
- 3 Verify the `.git` folder is gone with `ls -aa` (*macOS/Linux*) or `dir /a` (*Windows*).

With the Git folder destroyed, we can now reinitialize our Git repo. Do you remember how it’s done? Here’s a bullet list to help you remember:

- Navigate to the folder you want to track (`cd ~/Dev/roadtok8s/py/`).
- Initialize the Git repo (`git init`).
- Create a `.gitignore` file and add files or folders you need to ignore.
- Add files or folders you want to track (`git add relative/path/to/file`).
- Commit your changes (`git commit -m "Commit message"`).
- Push your code when ready. If you deleted your Git history, you might have to append `--force` to your pushes like `git push origin main --force` to *force* the changes to be pushed to the remote repository. Forcing the push with `--force` will overwrite any changes that may have been made to the remote repository and should be used with caution.

I encourage you to practice this process of destroying and recreating your Git repo as a way to practice Git and get comfortable with the process of tracking files. Before we push our code into a remote repo such as GitHub, let's review another important feature of Git: `git log`.

2.3.5 *Git logs and show*

Rebooting a Git repo is a great way to practice Git and get comfortable with the process of tracking files. However, it does have a downside: we lose our commit history. This section will help us better understand how we can revert to previous versions of our code, so we might have to reboot our repos less and less.

One of the reasons for writing good commit messages is to help us understand what we did in the past and potentially revert to a previous state. To do this, we can review our commit history using `git log`. See the following listing.

Listing 2.29 Reviewing our Python commit history log

```
cd ~/Dev/roadtok8s/py/  
git log
```

If we are working in a repo that has a commit history, our output should resemble figure 2.9.

```
commit 278484deaddcaf231e099b5684734f87f0f7366b (HEAD -> main)  
Author: Justin Mitchel <my-email>  
Date:   Wed Nov 23 15:50:28 2022 -0600  
  
    Initial Python commit
```

Figure 2.9 Initial Git commit log for our Python project

Within this single log entry, we see a few important pieces of information:

- *commit <commit-id> (HEAD -> main)*—This is the commit id for the branch we are on (e.g., main or possibly master).
- *Author*—This is the configured author for this commit. This is great for teams.
- *Date*—This is the date and time of the commit itself. This is another feature that is great for personal and team use so we can verify when a change was made.
- *Commit message*—Finally, we see our commit message (e.g., "Initial Python commit"). Even from the first message, we can see how what we write here can help inform what happened at this commit event to help us dig deeper.

As we can see, the commit message can play a crucial role in reviewing any given point in our history. Better commit messages are better for everyone, including a future

version of you! Our commit id (e.g., “278484deaddcaf231e099b5684734f87f0f7366b”) can be reviewed by entering the command `git show <commit-id>` as shown in the following listing.

Listing 2.30 Git show example

```
git show 278484deaddcaf231e099b5684734f87f0f7366b
```

This will give me all of the files I added to the commit with the changes associated, as seen in figure 2.10.

```
diff
new file mode 100644
index 0000000..0c51dbc
--- /dev/null
+++ b/src/main.py
@@ -0,0 +1,20 @@
+import pathlib
+from fastapi import FastAPI, Request
+from fastapi.responses import HTMLResponse
+from fastapi.templating import Jinja2Templates
```

Figure 2.10 Git show example

Even if this is a bit difficult to read, let’s let it sink in for a moment: we see exactly what has changed with any given file at any given point in time. The + sign indicates a line that was added, and the - sign indicates a line that was removed (which is not shown in this particular commit). This is incredibly powerful and can really help us build better more maintainable code.

If we need to revert back to this point in time, we can with the `git reset <commit-id>` command. `git reset` (and `git reset <commit-id> --force`) should be used sparingly because they can literally *erase a lot of your hard work* without using other features of Git (e.g., branches).

If you need to revert to older versions of your code, I always recommend storing your code on another computer (called a remote repository) so you can always get back to it. We’ll cover this in the next section. Before we push our code elsewhere, let’s review a few of the fundamental `git` commands.

2.3.6 Fundamental Git commands overview

It is absolutely true that Git can become very complex, so I want to show you a few key commands for your reference:

- `git init`—Initializes a Git repository.
- `git add <relative/path/to/file>`—Adds a file to be tracked.
- `git add <relative/path/to/folder>`—Adds a folder to be tracked.

- `git commit -m "Commit message"`—Commits or finalizes changes with a custom message.
- `git push`—Pushes changes to a remote repository.
- `git pull`—Pulls changes from a remote repository.
- `git log`—Shows all recent commits.
- `git show <commit-id>`—Shows all files and changes associated with a commit.
- `git reset <commit-id>`—Reverts to a previous commit. Omit `<commit-id>`, and it will revert to the most recent commit.
- `git reset <commit-id> --hard`—Reverts to a previous commit and erases all changes since then. (Use this with caution).

Learning and understanding Git takes time and practice, so I encourage you to review these commands as frequently as possible. Now that we can track our code, let's push it to a remote repository so it can be hosted away from our machine and potentially shared with others.

2.4 **Pushing code to GitHub**

GitHub.com has been a longtime friend and pioneer of the software development community. GitHub hosts code for free with both private and public modes for millions of developers worldwide for free. In this section, we will configure our GitHub account to host the two projects we created in this chapter.

We'll keep this part hyper-focused on pushing code to GitHub because we'll continue to use more and more features of GitHub throughout this book.

Here are the steps we'll take:

- 1 Create a GitHub account.
- 2 Create a new repository for each of our applications.
- 3 Push our code to GitHub.

This section is really about implementing the `git push` command into our workflow. Let's create a GitHub account if you do not already have one.

2.4.1 **Creating a GitHub account**

For the purposes of this book, you will need to sign up for GitHub. There are alternatives to GitHub, but this book has been designed around storing code on GitHub as well as using automation pipelines with GitHub Actions in later chapters. The alternatives to GitHub, such as GitLab, are perfectly capable of performing these activities, but configuring them is outside the scope of this book.

In the next chapter, we will see how to deploy code by using our self-hosted repo without GitHub, so you can see from a first-principles perspective how Git-related hosting works.

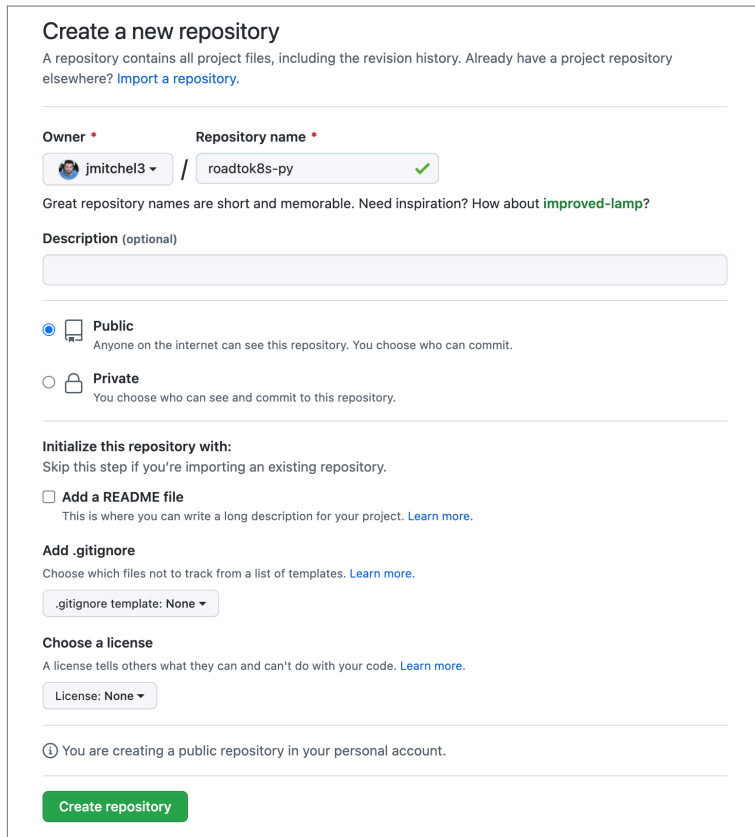
For now, create an account on GitHub at <https://github.com/signup>. Once you have a GitHub account, let's create a couple of repositories to host our code.

2.4.2 Create a Git repository on GitHub

We're going to create two new repositories on GitHub, one for each of our applications. One repo per application is highly recommended; otherwise, your projects start to get far too complex.

Create your first repo by going to <http://github.com/new>. If this is the first time you've created a repository on GitHub, this link might become your new best friend.

Start with your Python app and the following configuration. Notice that I left a lot of things blank or unchecked. This is intentional since we already configured a local repository (figure 2.11).



The screenshot shows the 'Create a new repository' page on GitHub. At the top, it says 'Create a new repository' and provides a brief explanation of what a repository is. Below this, there are two input fields: 'Owner' and 'Repository name'. The 'Owner' field is set to 'jmitchel3' and the 'Repository name' field is set to 'roadtok8s-py'. A green checkmark is visible next to the repository name. Below these fields, there is a note about repository names being short and memorable, with a suggestion for 'improved-lamp'. A 'Description (optional)' text area is present but empty. Under the 'Public' and 'Private' radio buttons, the 'Public' option is selected. Below this, there is a section for 'Initialize this repository with:' which includes an unchecked checkbox for 'Add a README file'. The 'Add .gitignore' section shows a dropdown menu set to '.gitignore template: None'. The 'Choose a license' section shows a dropdown menu set to 'License: None'. At the bottom, there is a note stating 'You are creating a public repository in your personal account.' and a green 'Create repository' button.

Figure 2.11 Creating a repository on GitHub

After you select Create Repository, you will be taken to the new repository's page. You will see a section called Quick Setup that will give you a few options for how to get

started with your new repository. We're going to use the . . . or push an existing repository from the command line option.

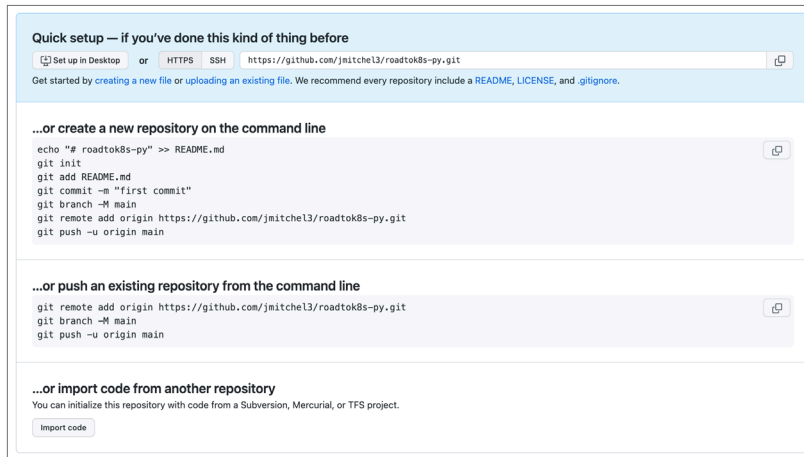


Figure 2.12 A newly minted repository on GitHub for our Python app

Before you get too excited and start copying and pasting code from GitHub, let's drill down on what needs to happen to configure our local Git repository to push to this GitHub repository.

2.4.3 *Configuring our local git repo for GitHub*

Before we can push our code to GitHub, we need to ensure our local Git repository is ready to push code to the GitHub repository. We will do this by adding a remote to our local Git repository. There's a key line in the Quick Setup section that we need to pay attention to (figure 2.13).

...or create a new repository on the command line

```
echo "# roadtok8s-py" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/jmitchel3/roadtok8s-py.git
git push -u origin main
```

Figure 2.13 `remote add` via new GitHub repo

Do you know why this is the key line? It's because we already ran all the other commands! With the exception of creating a Readme.md file, we ran the following commands in the previous section:

- `git init`

- `git add .`
- `git commit -m "Initial commit"`

We will have to continue to use `git add path/to/some/file/or/folder/` along with `git commit -m "Commit message"`, but we will not have to run `git init` or `git remote add` again.

Let's run the `git remote add` by navigating into your project `cd ~/dev/roadtok8s/py/` and running `git remote add origin https://github.com/yourusername/roadtok8s-py.git` (replace `yourusername` with your GitHub username) as GitHub recommends. `origin` is the name we are giving to the remote repository because `origin` is a conventional name for your primary remote repository. In the next chapter, we'll revisit this concept with a different name for a remote repository.

Now that we have added the remote repository, we can push our code to GitHub, as shown in the following listing.

Listing 2.31 GitHub push

```
git push origin main
```

Did you have errors?

- If you run into an error with `main` not being a valid branch, try `git branch --show-current` to see which branch you're using.
- If you run into an error with `origin` not being a valid remote, try `git remote -v` to see which remotes you have configured.

Did you succeed? If so, congratulations! You have just pushed your project's code to GitHub! We are now one step closer to pushing our code into production. Now it's time to do the same with our other project.

2.4.4 Rinse and repeat

Repeating the process for our Node.js project should be simple at this time, but as a recap, here's what you need to do:

- 1 Create a new repository on GitHub with just the name `roadtok8s-js`.
- 2 Copy the `git remote add` command from GitHub.
- 3 Add the remote repository to your local repository `cd ~/dev/roadtok8s/js/ && git remote add origin https://github.com/yourusername/roadtok8s-js.git`.
- 4 Push your code to GitHub with `git push origin main`.

Using Git is crucial for building maintainable applications as well as ones that we can continuously deploy into production. This book will continue to build on the concepts of Git and GitHub as we move forward.

Summary

- Python and JavaScript are both excellent choices for learning the basics of building web applications.
- Even simple applications have a lot of dependencies and moving parts to get up and running.
- Using third-party package tracking with `requirements.txt` for Python and `package.json` for JavaScript/Node.js helps to create a repeatable and consistent environment for your application.
- FastAPI can be used to create a simple web application in Python that can handle the request/response cycle.
- Express.js can be used to create a simple web application in JavaScript/Node.js that can handle the request-response cycle.
- Template engines can be used to render HTML content in Python and JavaScript/Node.js, thus eliminating repetitive code.
- Functions can be used to map to URLs while handling HTTP methods in FastAPI and Express.js.
- Git is a version control system that can be used to track changes to your code and revert to previous versions of your code.
- Using version control is an essential first step in building robust applications that are deployable, shareable, and maintainable.

Manual deployment with virtual machines

This chapter covers

- Creating and connecting to cloud-based servers
- Understanding web server fundamentals with NGINX
- Installing self-hosted Git repositories
- Configuring and installing production web applications
- Implementing background-running applications with Supervisor
- Automating the deployment process with Git hooks
- Securing our server by leveraging firewalls and SSH keys

In this part of our journey, we will manually deploy our code to a server. This involves uploading our code from our local computer to our server using just Git. Once the code is uploaded, custom scripts will be triggered to ensure that our server is updated with the latest version of our application. The process of sharing code safely to and from computers is exactly why Git is such an important tool and well worth mastering.

This process is similar to driving a vehicle with a manual transmission. Although it requires a bit more work, it's an excellent opportunity to develop your skills before moving on to the automatic method. If you're from the United States, you may be accustomed to driving an automatic transmission and find this idea strange. However, if you know how to drive a manual transmission, you might have a great appreciation of the value of learning the manual process before the automatic one.

Learning how to deploy manually will teach us the process and provide us with a desired outcome while only being slightly more challenging than writing application code like we did in chapter 2. Platform-as-a-service (PaaS) providers, such as Heroku and AWS Elastic Beanstalk, are tools that use application code and Git to automate many of the steps we'll do manually in this chapter. PaaS providers can be a great way to deploy an application without worrying about the underlying infrastructure. The tradeoff in using a PaaS often comes down to price and flexibility.

To succeed in this chapter, you should have a basic understanding of the HTTP request/response cycle, URL functionality, and some command-line experience. Additionally, you should have completed two web applications in chapter 2.

Our first step is to create a remote server to deploy our code to. We will use a cloud provider called Akamai Connected Cloud (ACC) to provision a virtual machine (VM) to host our code. We will use Ubuntu 22.04 LTS as our operating system.

3.1 *Creating and connecting to a remote server*

Cloud computing has unlocked a massive barrier to entry when it comes to deploying an application at nearly any scale. If cloud computing didn't exist, you would need to buy all the computers (e.g., physical servers) and related networking equipment yourself. You would also have to negotiate with the internet service providers (ISPs) to get your own static IP addresses, among many other challenging obstacles that are beyond the scope of any one book. Static IP addresses are how we can have our websites and apps run on the internet. Cloud providers handle the physical computing infrastructure for us: the servers, the networking, the cables, the power, the cooling, the physical security, the IP addresses, and many other things to ensure our software can run. That's where ACC (formally Linode) comes in.

ACC is a leading global cloud provider that offers a wide range of cloud services, including VMs, content delivery networks, object storage, databases, and managed Kubernetes. We will use ACC as our primary cloud provider for this book, but the concepts and techniques can be applied to nearly any cloud provider thanks to the power of the open-source technologies we have been and will continue to use.

A key theme of this book is portability and flexibility. This chapter is no different. The goal is to have the skills, techniques, and understanding of how we can deploy our applications to nearly any server, hosting service, or cloud provider.

3.1.1 Provisioning a virtual machine

Provisioning a VM is the process of renting a portion of a server or servers from a cloud provider. The physical server utilizes virtualization, allowing you to rent only a distinct portion of a larger computer rather than the entire machine. If you think of it as a shopping center, you only have to rent one unit instead of the entire complex. One significant advantage of this rental process is the ability to select the desired amount of compute power, including RAM, CPUs, and more, resulting in greater efficiency and cost savings. This paradigm forms the foundation of cloud computing, which, as we will learn in this book, is an excellent method for deploying all kinds of applications.

The specific type of VM we are going to rent is provided by ACC's Linode service. Let's do that now:

- 1 Create an account on ACC at linode.com/rk8s (book readers receive a promotional credit as well).
- 2 Log in to ACC (<https://linode.com>).
- 3 Click the dropdown Create > and then select Linode.
- 4 Use the following settings:
 - *Distribution*—Ubuntu 22.04 LTS (or latest LTS)
 - *Region*—Dallas, TX (or the region nearest you)
 - *Plan*—Shared CPU > Linode 2 GB (or larger)
 - *Label*—rk8s-vm
 - *Tags*—<optional>
 - *Root Password*—Set a good one
 - *SSH Keys*—<recommended> (review appendix D to learn how to create and one and add it to Linode)
 - *All other settings*—<optional>
- 5 Click Create Linode.
- 6 After a few minutes, you will see an IP address appear in the Linode Console (figure 3.1).

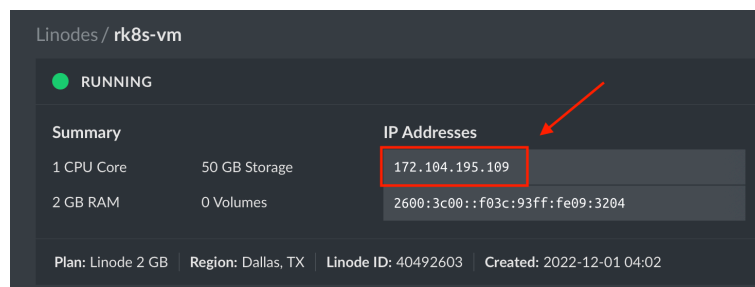


Figure 3.1 Linode VM provisioned with an IP address

If this is your first experience with provisioning a VM or remote server, it's a good time to let the significance of what you just accomplished set in. With under \$10 (or probably \$0), you rented a portion of a computer located in a data center in Dallas, Texas, or another location of your choice. You now have complete control over this computer and can run any type of application on it. Additionally, you have an IP address accessible from anywhere in the world, allowing you to positively affect someone's life on the opposite side of the globe with your application. Best of all, it only took around 5 minutes to get this server up and running.

This same process is what the big tech companies use every day: provisioning VMs to run their applications. The biggest difference between our needs in this book and theirs often comes down to the sheer volume of VMs and the power we may need. For example, we may only need one VM to run our application, while they may need thousands. The good news is that the process is the same and can be improved using techniques called *infrastructure as code* (IaC). IaC is outside the scope of this book but is well worth studying if you are looking for a better way than manually provisioning servers through a web console (this process is often referred to as ClickOps).

Now that we have a remote server ready, we need a way to start controlling it. For this, we will use a protocol known as *Secure Shell (SSH)*, which is a secure way to log in and connect to our VM. Once connected, we can start configuring it and running all kinds of commands. SSH is how we access the server's command line to do things like install server-based software, work with application dependencies, configure a bare Git repository, and ultimately deploy our application. (IaC can also do wonders for this process as well.)

3.1.2 **Connecting via SSH**

Connecting to a remote server is so fundamental these days that most desktop and laptop computers have a built-in SSH client. Mobile devices have varying support natively, but almost all of them have apps that support SSH. SSH can be used to connect to computers on our local network as well as computers across the world through the internet. To perform SSH, we need at least five things:

- A remote host
- An IP address or domain name (figure 3.2)
- A username
- A password or an installed SSH key
- A connection to the same network as the remote host (e.g., the internet)

Four of these five things were completed when we provisioned our VM. I'll go ahead and assume you are connected to the internet to access our remote host on Akamai Linode.

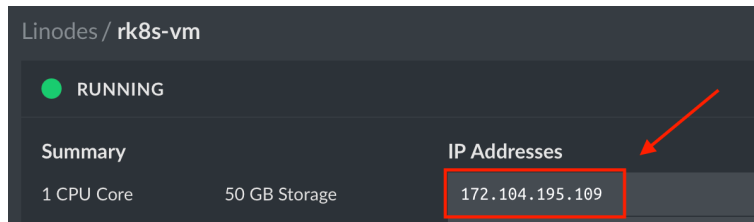


Figure 3.2 Our current host IP address

Let's see how these items map to what we'll use:

- *Remote host*—Akamai Linode virtual machine
- *IP Address*—172.104.195.109 (see figure 3.2)
- *Password*—Root password (see section 3.1.1)
- *Username*—root

The username of `root` is the default for many Linux distributions and Ubuntu operating systems. Some cloud providers change this username, and it may change on Akamai Linode in the future as well. In our case, we can find this information within the Akamai Linode console, as seen in figure 3.3.

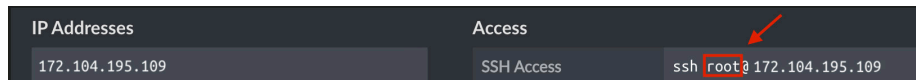


Figure 3.3 Akamai console SSH connection example

Now that we have all the information we need, let's connect to our remote host. Depending on your configuration, you may be prompted for your root user password. With this in mind, open your command line (Terminal or PowerShell) and enter the command shown in the following listing.

Listing 3.1 SSH connection example

```
ssh root@172.104.195.109
```

Since this is our first time connecting to this remote host and this particular IP address, a few standard processes will occur. The first process is a prompt, like in figure 3.4, that is asking if we want to allow our local computer to trust and connect to the remote server. This authenticity check should only happen one time *unless* something has changed on the remote host or our local machine's records of remote hosts.

```
The authenticity of host '172.104.195.109 (172.104.195.109)' can't be established.  
ED25519 key fingerprint is SHA256:QQ0Gq6tTS1N1CxJ/ytqX4vuLREpozQQ1bxBP4bVrIQ.  
This key is not known by any other names  
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Figure 3.4 Verifying the authenticity of host confirmation

If you see this, type `yes` to continue and skip to the next section. After you confirm with `yes`, this message will not appear again unless you connect with a new computer or you remove the host IP address (e.g. `172.104.195.109`) from your `~/.ssh/known_hosts` file. If you do not see this response, there are a few possible reasons:

- You used the wrong username or IP address in your `ssh` command.
- Your SSH key is invalid or incorrectly installed (learn how to use SSH keys in appendix D).
- You or the server have a firewall blocking the connection, which is possible if this is *not* a new server.
- You have used SSH for this particular IP address, so you'll need to remove it from `~/.ssh/known_hosts`.

After we verify the authenticity of the remote host, we will be prompted for the root user password. You will be prompted for this password every time you connect to this remote host unless you have an SSH key installed on the remote host. Installing an SSH key on a remote host for password-less connect with SSH is covered in appendix C. At this point, your local command line should become the remote command line that resembles figure 3.5.

```
The programs included with the Ubuntu system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*/copyright.  
  
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by  
applicable law.  
  
root@localhost:~#
```

Figure 3.5 Remote host command line first entry

If you see this output, you have successfully connected to your remote host. For the second time in this chapter, it's time to celebrate because your computer is now connected directly to another computer over the internet without using a web browser! Congratulations!

It might not surprise you to learn that your first step will be to update your system. We are not *upgrading* the operating system but rather updating our system's list of available packages.

Before we run the command to update, let's learn about what `sudo` and `apt` are. `sudo` is a command that allows you to run a command as the superuser with special permissions to execute or install something. `sudo` should be used sparingly, but it is often required when we make changes to the software or configuration of the system. `apt` is a package manager for Debian-based Linux distributions. You can consider this command similar to `npm` or `pip`, because it installs all kinds of third-party software on our behalf but specifically for Linux systems. `apt` can also be used to install various programming language runtimes like Python and Node.js, as shown in the following listing.

Listing 3.2 Using `apt` to update the system's list of available packages

```
sudo apt update
```

← This is a common command we will use when we start working with containers in future chapters.

We now have the perfect combination of ingredients to create our first static website on our remote host. It will be incredibly simple and use an amazing tool called NGINX (pronounced “engine-x”).

3.2 Serving static websites with NGINX

If we dialed the clock back about 20 years, we would find a lot of simple static websites that provide little more than an internet-based paper brochure. While simple static websites aren't as flashy as dynamic ones, they still serve a valuable purpose and can be used to great effect.

In this section, we're going to deploy our first website by merely modifying some HTML code and configuring a web server called *NGINX*. NGINX can do a great many things that we'll explore in future chapters, but for now, we're just going to use it to deliver a simple static website to our remote server's public IP address. Here is the process of how we will use NGINX:

- 1 Update the system's list of available packages.
- 2 Install NGINX.
- 3 Create a simple HTML file.
- 4 Configure NGINX to serve our HTML file.

Every time you install software on a Linux computer that is distributed through the APT Package Manager (`apt`), you should always update the list of available packages. This is because the list of available packages is updated frequently, and we want to ensure we are installing the latest version of the software. After we update this list, we will install NGINX with the following listing's command.

Listing 3.3 Installing NGINX on Ubuntu

```
sudo apt update
sudo apt install nginx
```

When prompted to install NGINX, type `y` and press Enter. This will install NGINX and all of its dependencies. A way to automate the `apt install <package>` command, is to include the `-y` flag so we can automatically agree to any prompts that might appear, ending up with a command like `sudo apt install nginx -y`.

Once NGINX finishes installing, you open your IP Address in your web browser (e.g., `http://172.104.195.109`). You should see something like figure 3.6.



Figure 3.6 NGINX installed and displaying the default page

Congratulations! You have successfully installed NGINX on your remote host and are now serving a static website to the world. We still need to configure this webpage to display our own content, but for now, this is exciting.

3.2.1 *Embrace ephemeral VMs*

When I was first deploying software like this, I got very attached to the VMs and the IP address I was using. I was afraid to make any big changes and “ruin” a lot of the work I did on that machine.

This fear was twofold: I was using an old method to manage files with a server (FTP), and I did not treat my VMs as if they were ephemeral.

File Transfer Protocol, or FTP, is still a valid method for moving files around on the internet, but I do not recommend it for code simply because Git exists. Git is far better because it allows you to track changes in code over time. FTP is more like the Wild West, and files might show up out of nowhere with very little explanation as to why. Personally, I would say avoid using FTP at all costs. If you *must* use something like FTP, you should consider using VSCode’s Remote-SSH extension to give you a more seamless experience.

The second reason I was afraid to make changes was that I did not treat my VMs as if they were ephemeral. Since I was using FTP, I got into this nasty habit of treating a VM like an external hard drive that happens to run my software as well. This is flawed logic because a VM is so easy to destroy and recreate that using it to persist data is a bad idea. Treating VMs as ephemeral will also help you prepare your code and projects for the correct tools to ensure that if you do need to destroy a VM, you can do so without losing any data.

With all this in mind, let's look at a few ways to remove NGINX and start over:

- *Destroy the VM*—This is the easiest way to start over and one I recommend you practice doing it.
- *Remove NGINX* via the APT package manager—This is a more common approach, because destroying and setting up VMs can get tedious, especially without automation and IaC tools:
 - `sudo apt remove nginx --purge` and `sudo apt autoremove` will remove NGINX and any other packages that are no longer needed.
 - `sudo apt purge nginx` and `sudo apt autoremove` will remove NGINX and any other packages that are no longer needed.
 - `sudo apt purge --auto-remove nginx` will remove NGINX and any other packages that are no longer needed.

Now that we have a way to install and remove NGINX, it's time we learn how to modify the default HTML page that NGINX serves.

3.2.2 Modify the default NGINX HTML page

To me, the NGINX default web page is such a joy to see. It's a reminder that I have started a new project that needs some configuring. To you, it might look like a website from the 1990s, and you would be exactly right. Either way, let's look at how we can slightly modify this page so we make it our own while learning about some of the basic concepts of NGINX.

After installing NGINX, our VM has a new path on our system at `/var/www/html`. This is the default root directory for serving static HTML files with NGINX. If we open this directory, we will see a file called `index.nginx-debian.html`. This is the default HTML page that NGINX serves. Let's change that page using our SSH connection and the command line. We'll start by deleting the default NGINX page with the following listing.

Listing 3.4 Removing the default NGINX page

```
sudo rm /var/www/html/index.nginx-debian.html
```

Let's create a new HTML file to serve as our new web page. To do this, we'll use a technique that uses `cat` (a command for displaying the contents of a file to the terminal), the `EOF` (End Of File) command, and a file path to create a new multiline file. The following listing shows an example of the syntax for this command.

Listing 3.5 `cat` EOF command syntax sample

```
cat <<EOF > path/to/your/file
This is a new line
This is another new line
    This is a tabbed line
This is the last line
EOF
```

Following this same syntax, let's see a practical example of using it by creating a new HTML file in the following listing.

Listing 3.6 Creating a new `index.html` file for NGINX

```
cat <<EOF > /var/www/html/index.html
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    <p>We're coming soon!</p>
  </body>
</html>
EOF
```

To verify the contents of this new file we can use `cat` once again with the command `cat /var/www/html/index.html`.

At this point, our `/var/www/html/` directory should just have `index.html` in it. Let's open up our IP address to review the changes on the actual internet, as seen in figure 3.7.

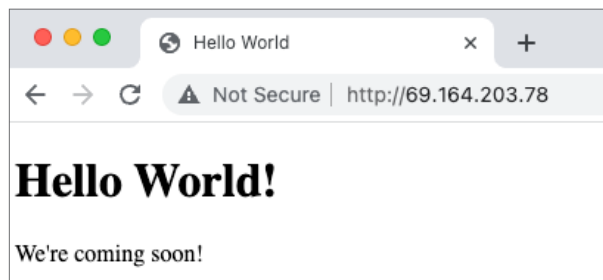


Figure 3.7 Verifying our new web page

As you may notice in figure 3.7, the IP address has changed from previous examples. Take a moment and ask yourself why you think that is. Was it a typo? A mistake in the book? No, it's simply that I performed the same exercise I asked you to do moments ago: destroying the VM and starting over. Throughout this book, we'll see IP addresses constantly changing because we continuously recycle our VMs.

It's important to note that we did not actually touch anything related to NGINX's configuration, but we did use the built-in features to serve a new HTML page. If you know frontend technologies well, you could improve the HTML we added significantly and make it look much better. For now, we'll leave it because it's time to set up a remote Git repository to store our application code.

3.3 Self-hosted remote Git repositories

The manual method of using `cat` and `EOF` is well suited for a quick and simple example but very poorly suited to managing a software project of any real substance. It's time we use our code from chapter 2 with Git to push (and pull) our code directly to a folder on our remote server.

Before we dive headfirst into using Git on this server, keep a few important ideas in mind when it comes to self-hosting a Git repository:

- **Data loss**—If we treat a VM as ephemeral, any code or data we store on that VM is not safe and will likely be permanently destroyed if our VM fails or is destroyed.
- **Little advantages**—Self-hosted Git repositories often offer very few benefits over using a third-party service like GitHub or GitLab. These tools have been refined for years to make managing Git projects much more effective than our own self-hosted solution.
- **Added and unnecessary complexity**—Although it may be simple to start and useful for basic examples, self-hosting Git repos can get convoluted quickly as our project scales.
- **Security**—Self-hosting Git repositories can be a security nightmare because collaborating on these repositories requires a multitude of steps to ensure the safety and security of the entire system.
- **Automation**—Git does not have a native way to run automated workflows within standard Git activities, whereas third-party tools like GitHub Actions and GitLab CI/CD can be used to automate the entire process of building, testing, and deploying code all through a single Git push.

With these items in mind, we still want to understand the mechanics of using a self-hosted Git repository as a means to transport our local code into production. In future chapters, we will transition this method toward hosting our code solely on GitHub and deploying through an automation pipeline.

In chapter 2, we configured two web applications that both had a local and private Git repository as well as a remote third-party Git repository hosted on GitHub (public or private). For the remainder of this section, we are going to configure one more location for our code for each project as a remote and private bare Git repository on our VM.

A *bare Git repository* is a repository that does not have a working directory, which just means that we have the history of changes but no actual code. This also means that we will not edit the code that lands on our server, but rather, we will only push and pull code from the bare repository. After our code exists in a bare repository, we will use a

Git-based hook to unpack our code into a working directory that will be the basis for our deployed application.

The end-to-end process of creating a remote and private bare repo is going to go like this:

- 1 Log into the remote server via SSH and install Git.
- 2 Create a bare Git repository for our Python and Node.js applications.
- 3 Update our local Git repository to point to the new remote repository, including any SSH keys as needed.
- 4 Push our code to the remote repository.
- 5 Create a Git hook that will allow us to
- 6 Check out our code into a working directory.
- 7 Install or update our app's dependencies.
- 8 Restart any web server processes.

3.3.1 *Log in to the server with SSH*

As our first step, we must be on the command line of our VM. Be sure to use SSH to log in right now, as in the following listing.

Listing 3.7 SSH into your host

```
ssh root@<your-ip>
```

Now that we are logged in, we can install Git on our server. In the following listing, we will use the same command we used in listing 3.8 to install Git on our server.

Listing 3.8 Installing Git on the server

```
apt-get update
apt-get install git -y
```

Now that we have Git installed, we can create our bare Git repositories.

3.3.2 *Creating the bare Git repositories*

Creating a bare repo is a matter of creating a directory and then running the simple Git command `git init --bare`. It's good practice to name these directories with a `.git` extension to help distinguish them from other directories and to make it clear that they are *bare* Git repositories.

After we create the bare Git repository, we need to create a symbolic reference with *HEAD* to the *main* branch. In this case, just like on GitHub, we'll use the *main* branch as our default branch. *HEAD* is just a way to identify the most recent commit in a repository. As we continue to make changes to our code locally, we want those changes to be reflected in our remote repository, and *HEAD* allows us to do this, as we'll see when we update the Git hooks in the next section.

Let's create the bare Git repos for both Python Web App and our Node.js Web App, as shown in the following listing.

Listing 3.9 Initializing the bare web app repos

```
mkdir -p /var/repos/roadtok8s/py.git
cd /var/repos/roadtok8s/py.git
git init --bare
git symbolic-ref HEAD refs/heads/main

mkdir -p /var/repos/roadtok8s/js.git
cd /var/repos/roadtok8s/js.git
git init --bare
git symbolic-ref HEAD refs/heads/main
```

Listing 3.9 highlights the same process we went through on GitHub (in chapter 2), except we used the command line instead of a website to create these new remote repositories. The format of the URL for these remote repositories will be as follows:

```
ssh://root@$VM_IP_ADDRESS/var/repos/roadtok8s/py.git
ssh://root@$VM_IP_ADDRESS/var/repos/roadtok8s/js.git
```

Replace `$VM_IP_ADDRESS` with the IP address of our current VM. To do this, we can use `curl ifconfig.me` or manually look it up in the Akamai Linode console as we did before. I prefer using `curl` as it's built-in with most Linux distributions, and it's far more automated.

Since we used the format `$VM_IP_ADDRESS` we can use built-in string substitution in bash to replace this value with the IP address of our VM. We can do this by running the command in the following listing.

Listing 3.10 Replacing the VM IP address

```
export VM_IP_ADDRESS=$(curl ifconfig.me)
echo "this is your IP: $VM_IP_ADDRESS"
```

To use the `VM_IP_ADDRESS` variable, we can use the following commands:

```
echo "git remote add vm ssh://root@$VM_IP_ADDRESS/var/repos/roadtok8s/py.git"
echo "git remote add vm ssh://root@$VM_IP_ADDRESS/var/repos/roadtok8s/js.git"
```

Your exact output from these commands will vary based on the IP address of your VM and thus is omitted from this book. Be sure to use your actual output before continuing.

Alarm bells should be going off in your head based on what we did in chapter 2 because now we can update our local code to point to these remote repositories, and we can even start pushing code here as well.

3.3.3 Pushing local code to the remote repo

Now that we have configured our remote *Git* hosts on our VMs, it's time to update our local project repositories to point to these new remote repositories. Once our local repositories are updated, we can push our code to the remote repositories.

The process goes like this:

- 1 Navigate to the root of our local project repository (`cd ~/Dev/roadtok8s/py/` or `cd ~/Dev/roadtok8s/js/`).
- 2 Add the remote repository to our local repository (`git add vm ..`).
- 3 Push our code to the remote repository (`git push`).

An example of this process for our Python Web App is as follows: `. cd ~/Dev/roadtok8s/py/ . git remote add vm ssh://root@45.79.47.168/var/repos/roadtok8s/py.git` (Replace 45.79.47.168 with your IP) `. git push vm main`.

In this example, we have used the remote named *vm* instead of *origin* for two reasons:

- *Origin*—This is already taken up by our GitHub remote repo as the *default* name for a remote repo.
- *vm*—Using this name signifies it is out of the norm for Git, and in future chapters, we'll move towards using just GitHub. This is also a signal that *vm* is a special case related to the lessons you are learning in this book.

Repeat this same process again for your Node.js. If you have a successful push to your VM, your output will resemble figure 3.8.

```
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (10/10), 1.02 KiB | 1.02 MiB/s, done.
Total 10 (delta 0), reused 0 (delta 0)
To ssh://
    4b825dc642cb6eb9a060e54bf8d69288fbee4904 -> main
```

Figure 3.8 Successful push to remote repo

If these pushes were not successful, you would see an error message. Here are a few error messages you may have encountered and what they mean:

- *fatal: 'vm' does not appear to be a Git repository*—Did you use `git remote add vm ssh://. . .`? If not, this is likely the error you will see.
- *fatal: Could not read from remote repository*—Do you have permission to access the remote repo? If not, this is likely the error you will see.

- *error: src refspec main does not match any*—This means your local branch is not *main* as we created in chapter 2. A quick fix can be as simple as `git branch -m master main` assuming that `master` is the branch you have been using.

If your local push was done correctly, our code should exist on our VM, but if you were to navigate into the bare repositories, you'll be met with some confusion as the code may appear to be nonexistent. Let's clarify this by checking out the contents of the bare repositories.

3.3.4 Git hook to check out code

The bare repositories are a special type of repository that only contains the metadata of the project. This means the code itself is not stored in the bare repository. Instead, the code is stored in a separate directory that is a clone of the bare repository. In our case, we have the metadata of the project already stored in the bare repository, but we need to clone the code from the bare repository to the directory where we will be running our code.

Now we will output the code to another directory that is a clone of the bare repository. This is done by using a *post-receive* Git hook. This hook is a bash shell script that is executed after a push to the remote repository. We will use this hook to clone the code from the bare repository to the directory where we will run our code.

First, let's create a new folder on our VM for each project, as shown in the following listing.

Listing 3.11 Creating a new folder for each project

```
mkdir -p /opt/projects/roadtok8s/py
mkdir -p /opt/projects/roadtok8s/js
```

The path `/opt/` is a common location for storing code on Linux-based systems. The path `/opt/projects/roadtok8s/` is where we will store our code for this book. The path `/opt/projects/roadtok8s/py/` is where we will store our Python code, and the path `/opt/projects/roadtok8s/js/` is where we will store our Node.js code.

Now that we have these two folders created, it's time to configure our Git hook to do the following:

- Clone the code from the bare repository to the directory where we will run our code.
- Install the dependencies for our project.
- Run or restart any running processes that allow our projects to run.

First, let's create the Git hook for our Python project. To do this, we will create a file named *post-receive* in the *hooks* directory of our bare repository. This file will be executed after a push to the remote repository. We will use this file to clone the code from the bare repository to the directory where we will run our code. After we create the file, we'll make it executable so Git will have permission to run the file, as shown in the following listing.

Listing 3.12 Creating the Git hook for our Python project

```
cd /var/repos/roadtok8s/py.git/hooks
touch post-receive
chmod +x post-receive
```

← This will make the file executable.

The filename *post-receive* is a special name for a Git hook so be sure to stick with this name otherwise the hook will not be executed. This file will contain the following:

- `git checkout`—This is the root command for checking out code with Git.
- `HEAD`—This is a reference specifying the latest commit on the branch of code we are checking out.
- `--work-tree=<path/to/working/dir/>`—This is a Git-specific flag that specifies the directory where the code will be checked out to.
- `--git-dir=<path/to/bare/repo/dir/>`—This is a Git-specific flag that specifies the directory where the bare repository is located. Using this flag allows our *post-receive* hook to be a bit more flexible.
- `-f`—This is a Git-specific flag that forces the checkout to overwrite any existing files in the working directory. This is useful if we want to update our code without having to manually delete the existing code.

With these concepts in mind, let's create the *post-receive* file with the command in the following listing.

Listing 3.13 Creating the *post-receive* file

```
export WORK_TREE=/opt/projects/roadtok8s/py
export GIT_DIR=/var/repos/roadtok8s/py.git

cat <<EOF > "$GIT_DIR/hooks/post-receive"
#!/bin/bash
git --work-tree=$WORK_TREE --git-dir=$GIT_DIR checkout HEAD -f
EOF

chmod +x "$GIT_DIR/hooks/post-receive"
```

Repeat this process for the Node.js project, replacing `py` with `js` wherever necessary.

Since the *post-receive* hook is just a bash script, we can call this hook at any time to ensure it's working properly. Any time you make changes to this hook, it's recommended that you call it immediately to ensure it's working properly, and code changes are being applied, as shown in the following listing.

Listing 3.14 Verifying the *post-receive* hook

```
export PY_GIT_DIR=/var/repos/roadtok8s/py.git
export JS_GIT_DIR=/var/repos/roadtok8s/js.git

bash $PY_GIT_DIR/hooks/post-receive
bash $JS_GIT_DIR/hooks/post-receive
```


If you see the error `error: pathspec 'HEAD' did not match any file(s) known to git`, there's a good chance you did *not* push your code from your local computer into your VM.

If you see an error or not, let's verify that we can push our code to our server right now with the code in the following listing.

Listing 3.15 Pushing code to server from local computer

```
cd ~/dev/roadtok8s/py
git push vm main

cd ~/dev/roadtok8s/js
git push vm main
```

Once that's pushed, you can verify the code on your server by running the following SSH commands:

```
ssh root@<your-ip> ls -la /opt/projects/roadtok8s/py
ssh root@<your-ip> ls -la /opt/projects/roadtok8s/js
```

At this point, we have all of the code on our machine, and we have a method for continuously installing and updating that code. The next step is to install the various software dependencies for our projects so they can actually run.

3.4 Installing the apps' dependencies

Earlier in this chapter, we saw how easy it can be to install NGINX to run a simple website. To have a more powerful and full-featured website, we are going to need to install a lot more software dependencies. More software dependencies mean more complexity, and more complexity means more things can go wrong. We will start to mitigate these potential problems by setting up specific environments for each project.

Since we have two different applications with two different runtimes, we will need to approach the installation of dependencies differently. For our Python project, we will use a virtual environment and the Python Package Tool (*pip*) with a specific version of Python 3. For our Node.js project, we will use the Node Version Manager (*nvm*) to install a specific version of Node.js and the corresponding Node Package Manager (*npm*).

The runtime requirements for our two projects are as follows. Verify each version with the commands next to it:

- *Python 3 version 3.8 or higher*—Verify this in your SSH session with `python3 --version`.
- *Node.js version 16.14 or higher*—Verify this with `node --version`.

Since we used *Ubuntu 20.04*, Python 3.8 should already be installed where Node.js is not. Regardless of what is currently installed, we are going to run through the process of installing different versions of Python and Node.js, so you're better prepared for long-term usage of these tools.

3.4.1 Installing Python 3 and our FastAPI project

The Python standard library has a lot of built-in features that make it ideally suited for web development. However, the standard library is not enough to build a full-featured web application, and some Linux-based machines do not have the full standard library of Python installed by default (unlike the macOS and Windows equivalents). To extend the capabilities of Python and prepare for a production-ready environment, we will need to install additional libraries on our Linux-based system because some of the standard library is *not available* on Linux machines without additional OS-level installations—for example, `venv` (<https://docs.python.org/3/library/venv.html>).

Python 3 is installed by default on Ubuntu systems; the newer the version of Ubuntu, the newer the default Python 3 installation will likely be. As always, we verify the version of Python on our system with `python3 --version` (or `python --version` for virtual environments) to ensure that our code can run with this version. Regardless of what version of Python 3 is installed currently, we are going to install the latest version along with a few other packages using the `apt` package manager:

- *python3*—This package provides the `python3` command, which is used to run Python 3.
- *python3-venv*—This package provides the `venv` module, which is used to create virtual environments.
- *python3-pip*—This package provides the `pip` tool, which is used to install Python packages.
- *build-essential*—This package provides the `gcc` tool, which is used to compile Python packages.

Each one of these packages is required to configure our Python application's environment and versions in a way that's suited for production use, as shown in the following listing. These packages will be installed once again when we start using containers with our Python application as well.

Listing 3.16 Standard Linux Python development packages

```
sudo apt update  ← Always run this before installing new packages.
sudo apt install python3 python3-pip python3-venv build-essential -y
```

Using `-y` will automatically answer yes to any prompts.

While installing, you may see a warning about Daemons using outdated libraries. If you do, hit *Enter* or *Return* to continue. Daemons are background processes that run on your machine that are updated from time to time, and this potential warning is an example of that.

After the Python installation completes, you should run `sudo system reboot` to restart your machine. This will ensure that all of the new packages are loaded and outdated daemons are refreshed. Running `sudo system reboot` will also end the *SSH* session, so you'll have to reconnect to continue. Rebooting after installs like this is

not always necessary, but since we changed global Python 3 settings, it's a good idea to reboot. Once our system is restarted, verify that Python 3 with `python3 --version` is installed and that the version is 3.10 or greater.

At this point, your mental alarm bells might be ringing because of the line *Python . . . version is 3.10 or greater*. This installation process will help us in deploying our basic Python application, but it leaves us with the question: Why did we not specify the exact version of Python we need? Installing specific versions of programming languages on VMs, including Python, is far outside the scope of this book. We actually solve this *versioning installation problem* directly when we start using containers later in this book. The reason that we are not installing a specific version of Python 3 is that there are far too many possible ways to install Python 3, and our code just needs Python 3.8 or higher to run in production.

It is now time to create a Python virtual environment to help isolate our Python project's software requirements from the system at large (see listing 3.17). To do this, we will create a dedicated place in the `/opt/venv` directory for our virtual environment. Removing the virtual environment from the project code will help us keep our project directory clean and organized, but it is ultimately optional, just as it is on your local machine. We will use the `/opt/venv` location time and time again throughout this book, as it is my preferred location for virtual environments on Linux-based systems as well as in Docker containers.

Listing 3.17 Creating the server-side virtual environment

```
python3 -m venv /opt/venv/
```

Going forward, we can use absolute paths to reference the available executables within our virtual environment. Here are a few examples of commands we will likely use in the future:

- `/opt/venv/bin/python`—This is the path to the Python 3 executable.
- `/opt/venv/bin/pip`—This is the path to the pip executable.
- `/opt/venv/bin/gunicorn`—This is the path to the gunicorn executable.

In production, using absolute paths will help mitigate problems that may occur because of system-wide Python installations as well as Python package version conflicts. What's more, if you use consistent paths and locations for your projects, you will end up writing less configuration code and get your projects running faster and more reliably. Absolute paths are key to a repeatable and successful production environment.

Now, we have the conditions to install our Python project's dependencies using *pip* from the virtual environment's Python 3 executable. We will use the *requirements.txt* file that we *pushed* and *checked out* with Git earlier in this chapter, located at `/opt/venv/roadtok8s/py/src/requirements.txt`. See the following listing.

Listing 3.18 Installing Python packages with absolute paths

```
/opt/venv/bin/python -m pip install -r \  
/opt/venv/roadtok8s/py/src/requirements.txt
```

We are almost ready to start running this application on the server, but before we do, I want to update my *post-receive* hook for Git so the command in listing 3.18 will run *every time* we push our code to our remote repository. This will ensure that our production environment is always up to date with the latest code and dependencies.

UPDATE THE GIT HOOK FOR THE PYTHON APP

Let's have a look at another practical use case for Git hooks: installing application software. To do this, we can use the command from listing 3.18 in our *post-receive* hook specifically because we used absolute paths to the Python executable and the *requirements.txt* file, as shown in the following listing.

Listing 3.19 Updating the Git hook for the Python app

```
export WORK_TREE=/opt/projects/roadtok8s/py  
export GIT_DIR=/var/repos/roadtok8s/py.git  
  
cat <<EOF > "$GIT_DIR/hooks/post-receive"  
#!/bin/bash  
git --work-tree=$WORK_TREE --git-dir=$GIT_DIR checkout HEAD -f  
  
# Install the Python Requirements  
/opt/venv/bin/python -m pip install -r $WORK_TREE/src/requirements.txt  
EOF
```

In this case, we overwrite the original *post-receive* hook with the contents of listing 3.19. If you want to verify this hook works, just run `bash /var/repos/roadtok8s/py.git/hooks/post-receive`, and you should see the output from the *pip* command. With all the dependencies installed for our Python application, we are ready to run the code!

RUN THE PYTHON APPLICATION

We have reached a great moment in our journey: we have a Python application that we can run on our server. It's time to run the application, see if it works, and verify we can access it from our browser. Both things should be true, but there's only one way to find out.

First, let's discuss the Python packages we need to run a production-ready Python application:

- *gunicorn*—Gunicorn is a Python-based *Web Server Gateway Interface (WSGI)* HTTP Server. In other words, gunicorn turns HTTP requests into Python code. Gunicorn is a great choice for production because it is a mature, stable, and fast HTTP server. Gunicorn is flexible, so you can use it with nearly any Python web framework: *FastAPI*, *Flask*, *Django*, *Bottle*, *Tornado*, *Sanic*, and many others. With gunicorn, you could even create your own Python web framework from scratch.

- *uvicorn*—Uvicorn is a Python-based *Asynchronous Gateway Interface (ASGI)* HTTP Server. ASGI is a Python specification for asynchronous HTTP servers. For local development, we can use *uvicorn* directly. When we go into production, we just use the *gunicorn* server with the *uvicorn* worker class.

The baseline configuration we'll need for *gunicorn* is the following:

- `--worker-class uvicorn.workers.UvicornWorker_`—This tells *gunicorn* to use the *uvicorn* worker class which is *required* for *FastAPI* applications (and potentially other ASGI applications).
- `--chdir /opt/projects/roadtok8s/py/src`—This tells *gunicorn* to change to the *src* directory before running the application. You *could* change the directory prior to running the application, but using the `--chdir` is my preferred option to follow.
- `main:app`—This tells *gunicorn* to look in *main.py* and find the *app* variable, because *app* is an instance of the `FastAPI()` application class. Because of the `chdir` flag, we don't need to specify the full path to the *main.py* file.
- `--bind "0.0.0.0:8888"`—This tells *gunicorn* two primary things: use *PORT 8888* on our local server and listen on *all* IP addresses. Binding *gunicorn* in this way means we *should* be able to access the application from our browser using our server's public IP address and port 8888.
- `--pid /var/run/roadtok8s-py.pid`—This tells *gunicorn* to write the process ID (PID) to the file `/var/run/roadtok8s-py.pid`. This is useful for stopping the application later on.

Let's now run this baseline configuration and see if it works, as shown in the following listing.

Listing 3.20 Command to run the Python application

```
/opt/venv/bin/gunicorn \
  --worker-class uvicorn.workers.UvicornWorker \
  --chdir /opt/projects/roadtok8s/py/src/ \
  main:app \
  --bind "0.0.0.0:8888" \
  --pid /var/run/roadtok8s-py.pid
```

Once you run this command, your application should start and return a domain of `http://0.0.0.0:8888`, which matches what we used to bind *gunicorn* to. If you have another application running on the specified port, this domain will not appear, but rather you will get an error. If all went well, you should see the same output as in figure 3.9.

```
[2023-03-08 21:35:58 +0000] [63003] [INFO] Starting gunicorn 20.1.0
[2023-03-08 21:35:58 +0000] [63003] [INFO] Listening at: http://0.0.0.0:8888 (63003)
```

Figure 3.9 Running the Python application output

You might be tempted to open your local browser to `http://0.0.0.0:8888`, but this will not work because this is running on cloud-based server. What we can do, thanks to the mapping of `0.0.0.0`, is open our browser to `http://<your-ip>:8888` and see the application running identically to your local version; see figure 3.10 for a live server example.

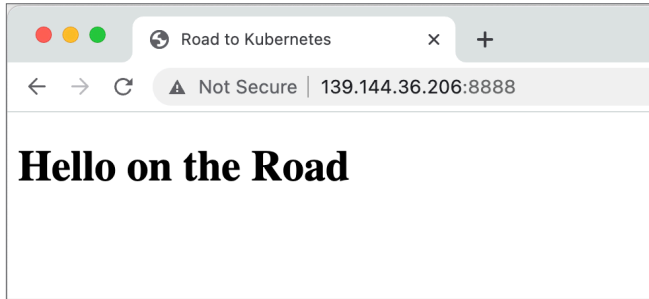


Figure 3.10 Python server running on port 8888

Congratulations! You have successfully run a Python application on a cloud-based server. We still have more to improve on this application, but we have a great foundation to build upon. Before we can continue, we must stop this server so we can make other changes to the server, including installing Node.js and our Express.js application.

To stop this application, we have three different options:

- Restart the VM (`sudo system reboot`).
- Use Ctrl+C on our keyboard as we would during local development.
- Use the process ID that is stored in the aforementioned `/var/run/roadtok8s-py.pid` path. We can run `kill -9 $(cat /var/run/roadtok8s-py.pid)` to stop the application with the contents of this file if it exists. *Do not remove* the PID file because that will not stop the application.

Now it's time to move over to our Node.js application. We'll install Node.js and our Express.js application on the same server as our Python application. Doing so makes implementation significantly easier but scaling significantly harder. Gracefully scaling these applications is one of the core reasons to adopt Kubernetes.

3.4.2 *Installing Node.js and our Express.js app*

Running multiple application runtimes on our machine is something we have now done twice with NGINX and Python 3. In this section, we'll configure our environment for our Node.js and Express.js application. Configuring environments can be tedious because each tool has a different set of dependencies to get running correctly, and then each application has a different set of requirements to get running, too. Node.js is no different.

Much like our local development environment, we'll need to install the latest LTS version of Node.js to ensure our application works correctly. The approach we'll take to do this is by using the Node Version Manager (nvm). *nvm* makes it incredibly simple to install various versions of Node.js and the associated Node Package Manager (npm).

Before we install *nvm*, your intuition might be to take the Linux-based approach using the *apt* package manager and the command `apt-get install nodejs`. While a *lot* of times this can be a suitable approach, just as we saw with Python, *apt-get* may install a version that's just too far out of date. At the time of writing this book, *apt-get install nodejs* installs a much older version of Node.js (version 12.22.9) than the one we need to safely run our application.

With that said, let's install *nvm* and the latest LTS version of Node.js using the Linux package *curl* along with an official *nvm* installation script.

INSTALLING THE NODE VERSION MANAGER

The Node Version Manager installation script and related documentation can be found at <http://nvm.sh>. Here's the process for how we'll install it:

- Declare a bash variable `NVM_VERSION` and set it to the version of *nvm* we want to install, and in this case, we'll use `v0.39.3`.
- `curl -o- <your-url>`—Using `curl -o-` will open a URL for us and output the response to the terminal. This is a bit like `echo 'my statement'` but for URLs. Using this will allow us to chain commands together with a pipe `|`.
- `bash`—We'll pipe the response from the URL and tell the command line (in this case, the *bash shell*) to run the script.

Let's see how this command looks in practice, as shown in the following listing.

Listing 3.21 Installing the Node Version Manager

Replace
v0.39.3
with the
version of
nvm you
want to
install.

```
export NVM_VERSION="v0.39.3"
curl -o- \
  https://raw.githubusercontent.com/nvm-sh/nvm/$NVM_VERSION/install.sh \
  | bash
```

After the installation finishes, your results should correspond to figure 3.11.

```
root@localhost:~# export NVM_VERSION="v0.39.3"
root@localhost:~# curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/
$NVM_VERSION/install.sh | bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
  Current                                 Dload  Upload   Total   Spent    Left
Speed
100 15916  100 15916    0     0  110k      0 --:--:-- --:--:-- --:--:--
110k
=> nvm is already installed in /root/.nvm, trying to update using git
=> Compressing and cleaning up git repository

=> nvm source string already in /root/.bashrc
=> bash_completion source string already in /root/.bashrc
=> Close and reopen your terminal to start using nvm or run the followin
g to use it now:

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion" # Th
is loads nvm bash_completion
```

Figure 3.11 Installing the Node Version Manager output

To use *nvm* right away, you need to either

- Close and reopen your SSH connection.
- Reload your bash profile with `source ~/.bashrc`.

Both of these ways are valid and will work, but I tend to reload my bash profile because it's a bit faster. The bash profile is essentially a file with a bunch of user-related configuration settings for commands and shortcuts.

With *nvm* installed and ready, let's verify it works and install the latest LTS version of Node.js (at the time of writing this book, v18.15.0) using *nvm install*.

Listing 3.22 Verifying *nvm* is installed

Option 1 for installing the latest LTS version of Node.js

```
nvm --version
nvm install --lts
nvm install 18.15.0
```

If this fails, try to exit your SSH session and reconnect.

Option 2 for installing a specific version of Node.js

As *nvm* completes, you should see the following output (figure 3.12).

```
root@localhost:~# nvm install --lts
Installing latest LTS version.
Downloading and installing node v18.15.0...
Downloading https://nodejs.org/dist/v18.15.0/node-v18.15.0-linux-x64.tar.xz...
#####
##### 100.0%
Computing checksum with sha256sum
Checksums matched!
Now using node v18.15.0 (npm v9.5.0)
```

Figure 3.12 Installing Node.js via *nvm* output

Assuming the *Node.js* installation via *nvm* was successful, your system should now have the following commands:

- *node*—The Node.js runtime; check the version installed with `node -version`.
- *npm*—The Node Package Manager; check the version installed with `npm -version`.
- *npx*—A utility for running Node.js packages; check the version installed with `npx -version`.

Now you have a repeatable way to install Node.js and the associated tools on your machine. This is a great way to ensure that you can install the same version of Node.js on any Linux machine you need to. *nvm* is supported on other operating systems, too; it's just not as easy to install as it is on Linux and Ubuntu. We can now finally get our Express.js dependencies installed and run our application.

INSTALL OUR APPLICATION DEPENDENCIES

As we discussed in chapter 2, we'll be using *npm* to install our application dependencies for our Express.js application. We'll start by navigating to our application directory

in `/opt/projects/roadtok8s/js/`, update the global version of npm, install our dependencies declared in `package.json` and then verify the dependencies were installed correctly with `npm list`, as shown in the following listing.

Listing 3.23 Installing Express.js app dependencies

```
cd /opt/projects/roadtok8s/js
npm install -g npm@latest
npm install
npm list
```

Just as before, this will create a `node_modules` directory in our application directory. With Node.js applications, we do not change the location of the `node_modules` like we did with the `venv` location for our Python project because Node.js applications do not require this step, for better or worse. We can now finally get our Express.js dependencies; let's update our `post-receive` hook to install them each time a new commit is pushed to our repository.

UPDATE THE GIT HOOK FOR THE JAVASCRIPT APP

Once again, we are tasked with updating our `post-receive` hook to install our application dependencies. In the following listing, we'll use the `npm` command to install our dependencies.

Listing 3.24 Updating the Git hook for the Node app

```
export WORK_TREE=/opt/projects/roadtok8s/js
export GIT_DIR=/var/repos/roadtok8s/js.git

cat <<EOF > "$GIT_DIR/hooks/post-receive"
#!/bin/bash
git --work-tree=$WORK_TREE --git-dir=$GIT_DIR checkout HEAD -f

# Install the Node.js Requirements
cd $WORK_TREE
npm install

EOF
```

In this case, we overwrite the original `post-receive` hook with the contents of listing 3.24. If you want to verify this hook works, run `bash /var/repos/roadtok8s/js.git/hooks/post-receive`, and you should see the output from the `npm install` command. With all the dependencies installed for our Express.js application, we are ready to run the code!

3.4.3 Running the Express.js application

In production environments, I prefer using absolute paths to the modules we intend to run. In our case, we will be running the `main.js` module in the `src` directory of our JavaScript application. We will also declare a `PORT` here as a way to ensure our application

is flexible to the PORT changing, as this will come in handy when we start shutting off ports in later sections. The command is as simple as the following listing.

Listing 3.25 Running the Node.js application

```
PORT=5000 node /opt/projects/roadtok8s/js/src/main.js
```

Now you should be able to open your browser at `http://<your-ip-address>:5000` and see figure 3.13.



Figure 3.13 Express.js Hello World

Congratulations! You have successfully installed Node.js and connected it to the outside world.

To stop this application, we have three different options:

- Restart the VM (`sudo system reboot`).
- Use Control+C on our keyboard as we would during local development.
- Use the process ID that is stored in the same directory as our `main.py` module at `/opt/projects/roadtok8s/js/src/main.pid`. This is a slightly different location than the Python-based one, but the concept is the same. Run `kill -9 $(cat /opt/projects/roadtok8s/js/src/main.pid)`.

At this point, we should have the ability to run two different runtimes for two different applications. The problem we face is ensuring these runtimes actually run without our manual input. To do this, we will use a tool called *Supervisor*.

3.5 *Run multiple applications in the background with Supervisor*

Supervisor is a simple way to turn our application run commands into *background processes* that will start, stop, or restart when our system does or when we need them to. *Supervisor* is certainly not the only way to do this, but it is rather approachable.

The setup process for any given application to be managed by *Supervisor* goes like this:

- Create a *Supervisor* configuration file for the application in `/etc/supervisor/conf.d/`.
- Update the *Supervisor* configuration file with the correct information for the application (e.g., working directory, command to run, logging locations, etc.).

- Update the git *post-receive* hook for various Supervisor-based commands to ensure the correct version of the application is running.

Let's start by installing Supervisor and creating a configuration file for our Python application.

3.5.1 Installing Supervisor

Installing Supervisor is just as easy as installing NGINX by using the *apt* package manager, as shown in the following listing.

Listing 3.26 Installing Supervisor on Ubuntu

```
sudo apt update
sudo apt install supervisor -y
```

Installing Supervisor will create a new folder for us located at `/etc/supervisor/conf.d/`. This is where we will store our configuration files for each application we want to manage with Supervisor.

Here are a few useful commands to remember when working with Supervisor:

- `sudo supervisorctl status`—List all of the applications currently being managed by Supervisor.
- `sudo supervisorctl update`—Update the configuration files for Supervisor. This is the same as `sudo supervisorctl reread && sudo supervisorctl update`.
- `sudo supervisorctl reread`—Read the configuration files for Supervisor.
- `sudo supervisorctl start <app-name>`—Start the application with the name *<app-name>*.
- `sudo supervisorctl stop <app-name>`—Stop the application with the name *<app-name>*.
- `sudo supervisorctl restart <app-name>`—Restart the application with the name *<app-name>*.

We use a few of these commands as we configure our applications.

3.5.2 Configure Supervisor for apps

To configure Supervisor for any given application, we'll need to do the following within a configuration file:

- Name the Supervisor process (`[program:yourname]`).
- Define the working directory for the application (`directory=/path/to/your/app`).
- Define the command to run the application (`command=/path/to/your/command`).

- Decide if the application should start automatically, restart automatically, and how many times it should retry to start. (`autostart=true`, `autorestart=true`, `startretries=3`).
- Define the logging location for the application in relation to `stdout` and `stderr`. (`stderr_logfile=/path/to/your/log`, `stdout_logfile=/path/to/your/log`).

Let's start with the Python application by creating a configuration file for it. Supervisor configuration is as shown in the following listing.

Listing 3.27 Supervisor configuration for Python

```
export APP_CMD="/opt/venv/bin/gunicorn \
  --worker-class uvicorn.workers.UvicornWorker \
  main:app --bind "0.0.0.0:8888" \
  --pid /var/run/roadtok8s-py.pid"
cat << EOF > /etc/supervisor/conf.d/roadtok8s-py.conf
[program:roadtok8s-py]
directory=/opt/projects/roadtok8s/py/src
command=$APP_CMD
autostart=true
autorestart=true
startretries=3
stderr_logfile=/var/log/supervisor/roadtok8s/py/stderr.log
stdout_logfile=/var/log/supervisor/roadtok8s/py/stdout.log
EOF
```

Looking at this configuration should be rather intuitive at this point. Let's create our Nodejs configuration file, as shown in the following listing.

Listing 3.28 Supervisor configuration for Node.js

```
export NODE_PATH=$(which node)
cat << EOF > /etc/supervisor/conf.d/roadtok8s-js.conf
[program:roadtok8s-js]
directory=/opt/projects/roadtok8s/js/src
command=$NODE_PATH main.js
autostart=true
autorestart=true
startretries=3
stderr_logfile=/var/log/supervisor/roadtok8s/js/stderr.log
stdout_logfile=/var/log/supervisor/roadtok8s/js/stdout.log
EOF
```

Gets the path to the Node.js executable based on your system's configuration

With both of these configurations in place, we need to create the directories for our output log files, as shown in the following listing.

Listing 3.29 Creating log directories

```
sudo mkdir -p /var/log/supervisor/roadtok8s/py
sudo mkdir -p /var/log/supervisor/roadtok8s/js
```

Now we have all the conditions necessary to run these applications via Supervisor. Let's run the following commands to update the Supervisor configuration and start the applications.

Listing 3.30 Updating the Supervisor Configuration

```
sudo supervisorctl update
```

Figure 3.14 shows the output from this command.

```
roadtok8s-js: added process group
roadtok8s-py: added process group
```

Figure 3.14 Supervisor processes added

Since we declared `autostart=true` in our configuration files, the applications should start automatically. Let's verify that they are running in the following listing.

Listing 3.31 Checking Supervisor status

```
sudo supervisorctl status
```

If we configured everything correctly, we should see the following (figure 3.15).

```
root@localhost:~# sudo supervisorctl status
roadtok8s-js      RUNNING    pid 89743, uptime 0:00:04
roadtok8s-py      RUNNING    pid 89738, uptime 0:00:09
root@localhost:~#
```

Figure 3.15 Supervisor status output

We should also be able to verify that our applications are running at

- `http://<your-ip>:8888` (Python via Gunicorn)
- `http://<your-ip>:3000` (Express via Node.js)

To start, stop, restart, or get the status of any Supervisor-managed application, it's as simple as: `sudo supervisorctl <verb> <app-name>` where *<app-name>* is either `roadtok8s-py` or `roadtok8s-js`. Here are a few examples:

- `sudo supervisorctl start roadtok8s-py`
- `sudo supervisorctl stop roadtok8s-py`
- `sudo supervisorctl restart roadtok8s-js`
- `sudo supervisorctl status`

The final step in this process is to update our post-receive hooks to restart either of these applications after the commit is pushed and the installations are complete, as shown in the following listings.

Listing 3.32 Supervisor post-receive hook

```
export WORK_TREE=/opt/projects/roadtok8s/py
export GIT_DIR=/var/repos/roadtok8s/py.git

cat <<EOF > "$GIT_DIR/hooks/post-receive"
#!/bin/bash
git --work-tree=$WORK_TREE --git-dir=$GIT_DIR checkout HEAD -f

# Install the Python Requirements
/opt/venv/bin/python -m pip install -r $WORK_TREE/src/requirements.txt

# Restart the Python Application in Supervisor
sudo supervisorctl restart roadtok8s-py
EOF
```

Listing 3.33. Supervisor post-receive hook update for Node.js

```
export WORK_TREE=/opt/projects/roadtok8s/js
export GIT_DIR=/var/repos/roadtok8s/js.git

cat <<EOF > "$GIT_DIR/hooks/post-receive"
#!/bin/bash
git --work-tree=$WORK_TREE --git-dir=$GIT_DIR checkout HEAD -f

# Install the Node.js Requirements
cd $WORK_TREE
npm install

# Restart the Node.js Application in Supervisor
sudo supervisorctl restart roadtok8s-js
EOF
```

Now that we have these applications always running, we need to start thinking about *how they can be accessed*. To do this, we'll implement NGINX as a reverse proxy to direct traffic to the correct application depending on the URL instead of using a port number. After we do that, we'll implement a firewall to limit access to this VM only to the ports we need.

3.6 **Serve multiple applications with NGINX and a firewall**

NGINX is much more than a tool to serve static websites; it can also be used to forward traffic *upward* to another application. In our case, this other application will exist on the same server as NGINX, but the application could, in theory, exist nearly anywhere as long as NGINX can reach it (e.g., through either a public or private IP address).

This process is called setting up a *reverse proxy*. The point of using reverse proxies is so we can redirect traffic, hide applications that may be running on a server (e.g., Python or Node.js), implement a load balancer, and much more.

Load balancing is merely the process of forwarding traffic to a server that can handle the traffic. If the server cannot handle the traffic, the load balancer will forward the traffic to another server if it can. Load balancing can get more complex than this, but that's the general idea. NGINX is designed to handle load balancing as well, but it's a concept we won't dive too deep into in this book.

3.6.1 Configuring NGINX as a reverse proxy

We want to configure NGINX so it forwards traffic to our *Supervisor-based* applications running at the following locations:

- `http://localhost:8888` (Python via gunicorn)
- `http://localhost:3000` (Express via Node.js)

It should be pointed out that these are localhost domain names (DNS), but they could easily be IP addresses or other public domain names.

We configured both applications to run on this server (localhost) and on the default ports for each application. We can configure NGINX to forward traffic to these applications by adding the following to our NGINX configuration file, as shown in the following listing.

Listing 3.34 NGINX reverse proxy configuration

```
cat <<EOF > /etc/nginx/sites-available/roadtok8s
server {
    listen 80;
    server_name localhost;

    location / {
        proxy_pass http://localhost:8888;
    }

    location /js/ {
        proxy_pass http://localhost:3000/;
    }
}
EOF
```

Location `/` will be the root index for this server that gets forwarded to the Python application.

Location `/js/` with the trailing slash will be forwarded to the Express.js application. If you omit the trailing slash, the Express.js application may not function properly.

The location `/etc/nginx/sites-available/roadtok8s` lets NGINX know this is a potential configuration file. We can enable this configuration by creating a symbolic link to the file in `/etc/nginx/sites-enabled/`. We do that by executing the code in the following listing.

Listing 3.35 Creating a symbolic link to the NGINX configuration file

```
sudo ln -s /etc/nginx/sites-available/roadtok8s \
    /etc/nginx/sites-enabled/roadtok8s
```

After we create this symbolic link, we need to remove the default configuration file that NGINX creates when installed by using `sudo rm /etc/nginx/sites-enabled/default,`

and then we'll restart NGINX with `sudo systemctl restart nginx`. `systemctl` is a built-in process manager, much like *Supervisor* that NGINX uses by default (`systemctl` is a bit more complex than *Supervisor* to configure, which is why we didn't use it earlier in the chapter).

Now open up your browser to the following locations:

- `http://<your-ip>`; will be served via Python and gunicorn.
- `http://<your-ip>/js` (including no trailing slash) will be served via Express via Node.js.

We now have a robust way to route traffic to our applications, but there's one glaring hole: *our PORTs are still accessible, too!* Let's fix this by installing a firewall.

3.6.2 **Installing Uncomplicated Firewall**

Adding a firewall to the VM is highly recommended because we want to deny all traffic that *is not* one of the following:

- HTTP traffic via port 80
- HTTPS traffic via port 443
- SSH traffic via port 22

Uncomplicated Firewall (UFW) is a simple and effective way to only allow for the previous PORTs to be available to the outside internet. In other words, UFW will automatically block access to any ports you do not need to keep open. We will block *all ports* except for the ones NGINX uses (80 and 443) and for our SSH (22) connections.

Let's install UFW:

```
sudo apt update
sudo apt install ufw -y
```

Whenever installing, update apt packages.

Install UFW if it's not already installed.

By default, UFW is disabled. Before we enable it, we need to configure it to allow for SSH and NGINX traffic.

UFW has a lot of great features that one might take advantage of, but for our purposes, we'll just allow for SSH and NGINX traffic. We want SSH traffic to be allowed so we can still access our VM via SSH and perform pushes via Git. We also want to allow all NGINX-based traffic to ensure our web applications are accessible. To do this, here are the commands we'll run

```
sudo ufw allow ssh
sudo ufw allow 'Nginx Full'
```

If you forget to allow SSH traffic, your current SSH session will end, and you may lose access forever. If this happens, you'll need to delete your VM and start over.

`Nginx Full` allows for both HTTP and HTTPS traffic, which maps to ports 80 and 443, respectively. We won't run HTTPs at this time but it's ready and available if we ever need to.

After running each of these commands, you will see the output `Rules Updated` or `Rules Updated (v6)` as a response.

Before we enable these changes, let's verify the current UFW configuration:

```
ufw show added
```

This should respond with

```
Added user rules (see 'ufw status' for running firewall):
ufw allow 22/tcp
ufw allow 'Nginx Full'
```

Now, we can enable our firewall:

```
sudo ufw enable
```

This will almost certainly respond with

```
Command may disrupt existing ssh connections. Proceed with operation (y|n)?
```

You should respond with `y` and press Enter with a result of `Firewall is active and enabled on system startup`.

If all went well, your firewall is now active, and your security has been improved. There are certainly other steps you might consider taking to further secure your VM, but those steps are outside the context of this book.

Now that we have a firewall in place, we can test our NGINX configuration by opening up a new browser window and navigating to `http://<your-ip>` and `http://<your-ip>/js`. You should see the same results as before.

The first time I ever deployed an application was using a variation of what we did in this chapter. It served me well for many *years* and served others well during that same time. I even had my database running on the same machine.

At this point, you should feel a little uneasy with the brittle nature of this setup. It *technically works*, but we have a major problem that needs to be addressed: if this machine goes down, the *entire production stack* does, too. In other words, this machine is a single point of failure, and here's why. The machine is

- Hosting the production Git repos
- Building and updating application dependencies and environments
- Running two primary applications in the background
- Hosting and running the NGINX web server

Here's a few things you might think of to help mitigate this problem:

- Upgrade this machine and make it more powerful.
- Back up this machine and its configuration so we can quickly provision another one if this one goes down.
- Create a replica of this machine and use a load balancer to distribute traffic between the two.

While these ideas might help, they do not solve the underlying problem. The best solution is to break up each component of our production stack into dedicated machines or services that can run independently of each other.

What's interesting is that we essentially configured a *remote development environment* that happens to have *production-like* qualities that the public can interact with. If we removed public access (via UFW) and just allowed SSH access, we would now have a powerful way to write our code from nearly any device using just SSH, a username, and a password.

Summary

- SSH is a powerful way to access and manage remote machines nearly anywhere in the world.
- NGINX is a tool that enables static website hosting, as well as more robust features like reverse proxies and load balancers.
- Installing and configuring software manually on a remote machine gives us a deeper insight into how to use the software and what the expected results are.
- Python and Node.js web applications are typically not run directly but rather through a web server like NGINX.
- Web applications should run as background processes to ensure they are always running with tools like *Supervisor*.
- Configuring a private host for our code is a great way to have redundant backups as well as a way to deploy code into production.
- Firewalls are an easy way to block specific ports from having unwanted access to our machine.

Deploying with GitHub Actions



This chapter covers

- Using a CI/CD pipeline
- Configuring and using GitHub Actions
- Enriching GitHub Actions workflows with secrets
- Document-based automation with Ansible

When you first learn to drive a car, everything is new, so every step must be carefully considered. After a few months or years, you start driving without thinking about every little detail. I believe this is the natural state for all humans—we learn and focus intently until it becomes automatic and effortless. Software is much the same. Our built-in human need to remove attention from any given task translates to how we deploy our software into production.

In chapter 3, we implemented elements of automation using Git hooks and Supervisor. Those elements are still important, but they’re missing something: automated repeatability. In other words, if we want to spin up a new server with the exact same configuration, we have to manually run through all of the same steps once again. To perform automated repeatability later in this chapter, we’ll use a popular Python-based command line tool called Ansible. Ansible uses a document-based approach to automation, which means we *declare* our desired state, and Ansible will *ensure* that state is met. This desired state would include things like NGINX is installed,

configured, and running; Supervisor is installed, configured, and running; and so on. Ansible is a very powerful tool that we'll only scratch the surface of in this chapter to discuss how it pertains to the tasks we're going to automate, continuing the work we began in chapter 3.

In addition to configuration automation through Ansible, we are also going to remove Git hooks from our workflow in favor of CI/CD pipelines. CI/CD pipelines can handle a few critical pieces for us, including running our configuration automation and testing our code to ensure it's even ready for live production. Further, we can use CI/CD pipelines to house our runtime environment variables and secrets (e.g., application-specific passwords, API keys, SSH keys, and so on) to be a single source of truth for our projects. This can be true regardless of how those variables and secrets are used (or obtained) in our code.

The value unlocked in this kind of automation comes down to the key phrase *portability* and *scalability*. Creating software that can move as you need is essential to maintain an adaptable project and keep all kinds of costs in check while still serving your users, customers, or whomever or whatever needs your software. We'll start by diving into our first CI/CD pipeline as a *Hello World* of sorts to show how simple it can be to manage software through Git-based CI/CD workflows.

4.1 **Getting started with CI/CD pipelines with GitHub Actions**

Pushing code into production is incredibly exciting. It's when the rubber hits the road and when you can make a major impact on the lives of the users you aim to serve. Or you could be releasing a dud. Either way, production is where the magic happens because even duds can teach us valuable lessons that are beyond the scope of software development.

Continuous integration, or CI, is the process of combining code from various sources (i.e., multiple developers, multiple machines) into a single project. Even if you are working alone, combining your new code with your old code on a regular basis is highly recommended, even if you completely wipe out your old code. Version control with Git makes continuous integration a simple process and gives us the confidence to make big changes to our codebase without fear of losing our work.

Whenever you merge or integrate code, it's highly recommended that you automatically test the code to ensure the code is stable and working as expected. There are numerous books on the subject of writing automated tests, also called unit tests, for any given programming language. So, while it's beyond the scope of this book, it's important to understand that automated tests are a critical component of continuous integration and thus continuous delivery or deployment.

Continuous delivery, or CD, is the process of automatically deploying code into an environment that is pre-production but production-like. This environment is often known as a staging environment. Its primary purpose is to have a manual review of the code as well as the result of any automated testing of the code. Once the staging environment is approved, the code is pushed into the final production stage, also known

as “live.” *Continuous deployment* is much like continuous delivery except that the code is automatically deployed into production without manual approval; it’s end-to-end automation with building, testing, and deploying to production.

For this book, we will perform CI/CD using a popular tool called GitHub Actions. GitHub Actions is a free service by GitHub.com that allows us to run code on their servers when we make changes to our stored repositories. GitHub Actions can be used to execute all kinds of code and run all kinds of automations, but we’ll use it primarily to deploy our applications and configure our environments. There are many CI/CD pipeline tools that exist to perform the exact actions we’ll do in GitHub Actions, including open source options like GitLab and nektos/act (see appendix D for how to use act).

4.1.1 Your first GitHub Actions workflow

GitHub Actions is a computer that runs code on your behalf based on your repo-level workflow-definition file(s). The process for it is as follows:

- 1 Create a GitHub.com repo for your project.
- 2 Create a workflow-definition file (or multiple files) in your repo following a specific file path and format (more on this shortly).
- 3 Commit the workflow file(s) and push your code to GitHub.
- 4 Run workflows on demand or automatically based on your workflow definition file(s).

This process is the same for many CI/CD tools out there; GitHub Actions is just one of many.

The workflows you define are up to you and can be as simple or as complex as you need them to be. They will run on GitHub’s servers and can run any code you tell them to run just as long as they can. How long you run code and how frequently are up to you. It’s good to keep in mind that although GitHub Actions has a generous free tier, the more you use it, the more it costs you.

For this book and many other projects, I have used GitHub Actions thousands of times and have yet to pay a cent; the same should be true for you. If using GitHub Actions is a concern, as it might be for some, we will also use the self-hosted and open-source alternative to GitHub Actions called nektos/act, which is almost a one-to-one drop-in replacement for GitHub Actions.

Now let’s build a *Hello World* workflow to get a sense of how to define and run workflows. In chapter 2, we created two GitHub repositories; for this section, I’ll use my roadtok8s-py repo (<https://github.com/jmitchel3/roadtok8s-py>).

In the root of your Python project, create the path `.github/workflows/`. This path is required for GitHub Actions to recognize your workflow definition files. For more advanced Git users, the workflows will only work on your *default branch* (e.g., *main* or *master*), so be sure to create these folders on your default branch.

Now that we have the base GitHub Actions Workflow folder created, we'll add a file named `hello-world.yaml` as our first workflow with the resulting path `.github/workflows/hello-world.yaml`. The workflow is going to define the following items:

- The name of the workflow
- When the workflow should run
- A single job to run (workflows can have many jobs)
- Which operating system to use (this is done via Docker container images)
- The steps, or code, to run in the job

We need to convert these items into YAML format because that is what GitHub Actions workflows require. YAML format is a human-readable data serialization format commonly used for configuration and *declarative programming*. YAML is often great for at-a-glance reviews because it's very easy to read, as you can see in the following listing.

Listing 4.1 YAML basics

```
name: Hello World
items:
  sub-item:
    name: Hello
    description: world
  sub-list:
    - element a
    - element b
  other-list: [1, 2, 3]
```

While listing 4.1 is just a simple example, you can see the format in action.

YAML is commonly used for *declarative* programming, which is concerned with the *output* and not the steps to get there. *Imperative programming* is concerned with the steps to get to the output. Python and JavaScript are both examples of *imperative programming* because we design how the data flows and what the application must do. YAML can be used for both kinds of programming paradigms.

Listing 4.2 is a YAML-formatted file that GitHub Actions workflows can understand. GitHub Actions is an example of declarative programming because we declare what we want to be done regardless of how exactly it gets done.

Listing 4.2 Hello World with GitHub Actions

```
name: Hello World
on:
  workflow_dispatch:
  push:

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
```

```
- uses: actions/checkout@v3
- name: Hello World
  run: echo "Hello World"
```

After you create this file locally, be sure to commit and push the file to GitHub with the following:

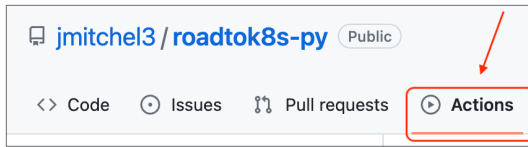
```
git add .github/workflows/hello-world.yaml
git commit -m "Create hello-world.yaml workflow"
git push origin main
```

To better understand what this workflow is doing, let's break it down:

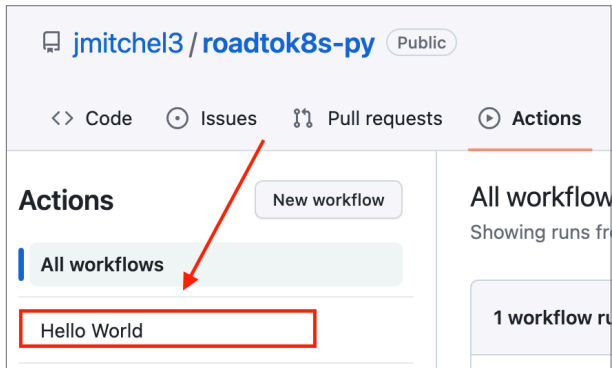
- **on:push**—This tells GitHub Actions to run this workflow every time we push code to our repository. We can narrow the scope of this configuration, but for now, we'll keep it simple.
- **on:workflow_dispatch**—This tells GitHub Actions that this workflow can be manually triggered from the user interface on GitHub.com. I recommend using this manual trigger frequently when learning GitHub Actions.
- **runs-on: ubuntu-latest**—Ubuntu is one of the most flexible options to use for GitHub Actions so we will stick with this option. There are other options for other operating systems but using them is beyond the scope of this book. Consider reading more about GitHub Action Runners in the official docs (<https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners>) and review the available images from GitHub (<https://github.com/actions/runner-images#available-images>).
- **steps**—This is where we define the steps, or code, we want to run. In our case, we only have one step, which is to run the command `echo "Hello World"`. This command will print the text *"Hello World"* to the console. Steps have several options including
 - **uses**—This is the name of the action we want to use (if it's not built into GitHub Actions). In our case, we're using the built-in action `actions/checkout@v3`, which is used to check out our code from GitHub. This is a required step for nearly all workflows.
 - **name**—We can optionally name each step; in this case, we just used *"Hello World"*.
 - **run**—This is the barebones command that allows us to execute any other command, such as `echo "my text"` or even `python3 my_script.py`.

Even in this simple workflow, we can see that GitHub Actions workflows can start to get rather complex rather quickly because they are so flexible. The flexibility is a great thing, but it can also be a bit overwhelming at first. The best way to learn GitHub Actions is to start with a simple workflow and then build on it as you need to.

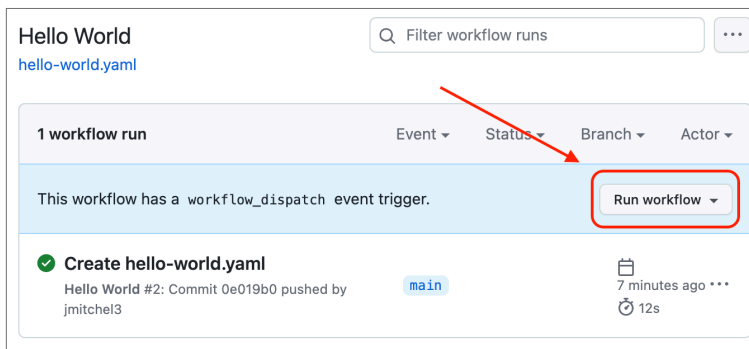
Let's review this workflow on GitHub.com by visiting the Actions tab of any given GitHub repository, as seen in figure 4.1.

**Figure 4.1** GitHub Actions tab

Once at this tab, you should see our GitHub Action listed in the sidebar, as seen in figure 4.2. If you do not see our workflow listed, you did not commit or push your code to the GitHub repository correctly, so you will have to go back and review the previous steps.

**Figure 4.2** GitHub Actions sidebar

The name listed here matches directly to the declaration of `name: Hello World` in the workflow file in listing << #github-actions-hello-world>>. To maintain the ordering of these names, it's often a good idea to put a number in front of the name (e.g., `name: 01 Hello World`). Now click `Hello World` and then click `Run Workflow`, as seen in figure 4.3.

**Figure 4.3** GitHub Actions Run Workflow

The reason we can even see Run Workflow is directly tied to the definition of `on: workflow_dispatch` in the workflow file.

Given that we included `on: push`, there's a good chance that this workflow has already run as denoted by `Create hello-world.yaml` in the history of this workflow, as seen in figure 4.3. Click any of the history items here to review the output of the workflow, as seen in figure 4.4.

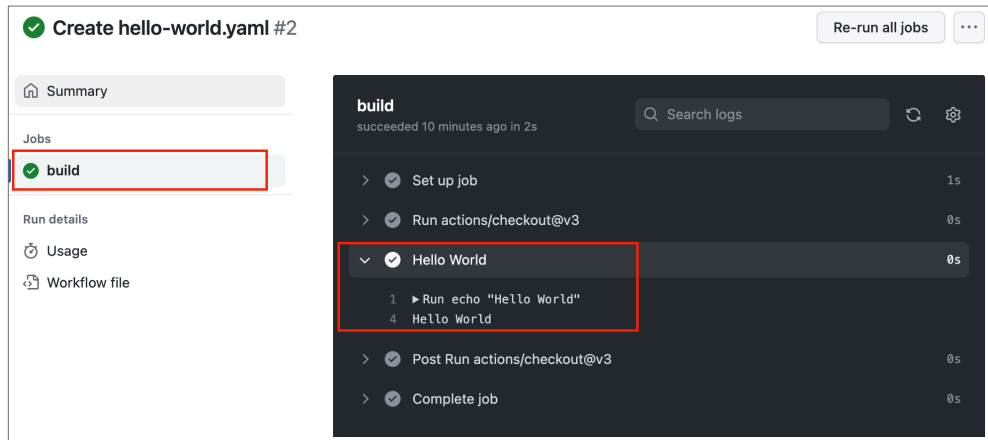


Figure 4.4 GitHub Actions workflow run result

Figure 4.4 shows a few important items:

- The name of the workflow that ran
- The jobs with the name `build` (again directly tied to the workflow file)
- The steps that ran in the job (including ones we did not define)
- The output of each step
- The output of our specific step named “*Hello World*”
- All checks passed (e.g., each icon for each step as well as the job have check icons instead of warning icons)

Congratulations! You just ran your first GitHub Actions workflow. This is a great first step into the world of CI/CD pipelines and GitHub Actions. Whether or not your workflow was successful is not the point. The point is you just used a serverless CI/CD pipeline to run code on your behalf. This is a very powerful concept we’ll use throughout this book.

GitHub Actions workflows are powerful indeed, but they become even more powerful when we give them permission to do a wider variety of things on our behalf. In the next section, we’ll look at how to give GitHub Actions workflows permission to do things like install software on a remote server by storing passwords, API keys, SSH keys, and other secrets.

4.1.2 Creating your first GitHub Actions secret

Now that we have our first workflow completed, let's start working toward a more practical example with the goal of simply installing NGINX on a remote server. This example requires us to use private and public SSH keys to work with GitHub Actions and our virtual machines at the same time.

We start by creating *purpose-built* SSH keys to be used specifically on GitHub Actions and Akamai Linode. The private key will be stored as a GitHub Actions secret, and the public key will be automatically installed on new instances provisioned on Akamai Linode.

Creating new SSH keys will ensure that our personal SSH keys are never exposed to GitHub Actions and thus are never exposed anywhere besides our local environment. Doing this will also set us up for even better automation for when we use Ansible with GitHub Actions later in this chapter. If you're new to working with SSH keys, consider reviewing appendix C. The following listing gives us the command to create a new SSH key without any input; it should work across different platforms.

Listing 4.3 Creating a new SSH key

```
# windows/mac/linux
ssh-keygen -t rsa -b 4096 -f github_actions_key -N ""
```

This command results in the following files in the directory you ran the command in

- `github_actions_key`—This is the private key we'll install in a GitHub Actions secret.
- `github_actions_key.pub`—This is the public key we'll install on Akamai Linode.

Once again, you should *never* expose or share your personal, private SSH keys (e.g., `~/.ssh/id_rsa`) because it poses a major security concern, which is exactly why we created a new SSH key pair for this purpose.

At this point, we will add the private key (e.g., `github_actions_key` without the `.pub`) to the GitHub repos we created in chapter 2. Here are the steps:

- 1 Navigate to your repo on GitHub.
- 2 Click *Settings*.
- 3 Navigate to *Secrets and Variables* on the sidebar.
- 4 Click *Actions*, as seen in figure 4.5.



Figure 4.5 GitHub Actions secrets link

At this point, click New Repository Secret and enter SSH_PRIVATE_KEY. Then paste the contents of your newly created private key (e.g., `github_actions_key`) into the value field. Click *Add Secret* to save the new secret. Figure 4.6 shows what this should look like.

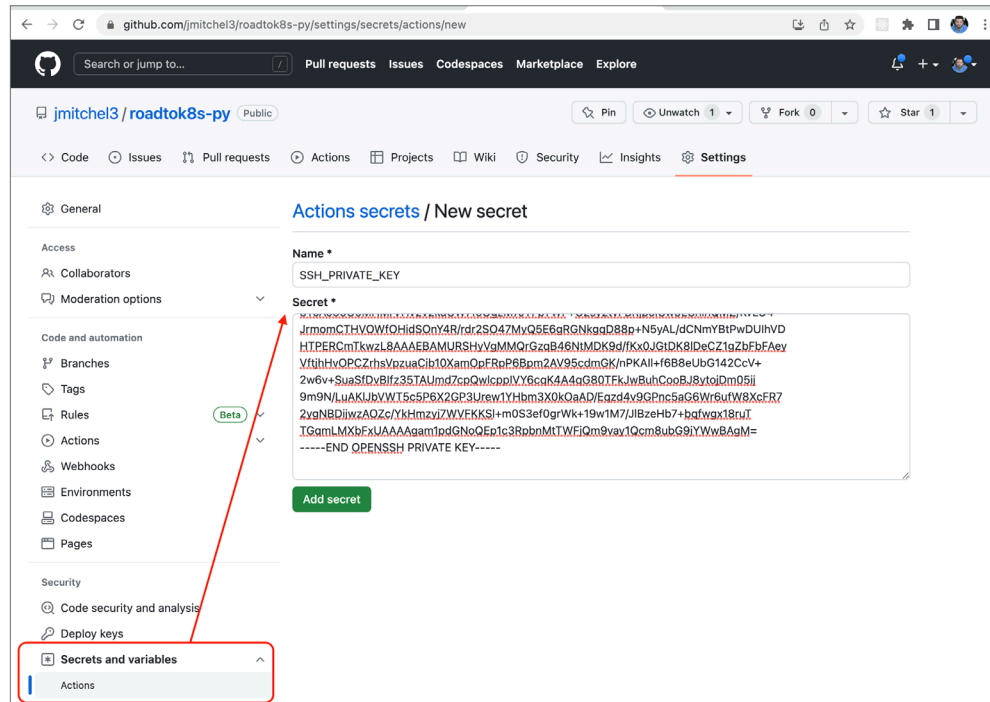


Figure 4.6 New GitHub Actions secret

Repeat this same process for any new secret you want to add. Figure 4.6 only highlights my Python application (<https://github.com/jmitchel3/roadtok8s-py>), but I recommend you repeat this exact same key on your Node.js application's repo as well (<https://github.com/jmitchel3/roadtok8s-js>). With the private key installed on GitHub Actions secrets, it's time to take the public key to the Akamai Connected Cloud Console.

4.1.3 Installing the public SSH key on Akamai Connected Cloud

Installing a public SSH key on our new instances provisioned by Akamai Connected Cloud will allow us to connect to the virtual machine without requiring a password or user input and enable GitHub Actions to perform the job(s) we need it to. Here are the steps to install a public SSH key to Akamai Connected Cloud:

1. Navigate to your Akamai Connected Cloud Console.
2. Click Account in the top-right corner.

- 3 Click SSH Keys on the sidebar, as seen in figure 4.7.
- 4 Click Add An SSH Key.
- 5 Add a label for your key (e.g., *GitHub Actions Public Key*) and paste the contents of your newly created public key (e.g., `github_actions_key.pub`) into the SSH Public Key field, as seen in figure 4.8.

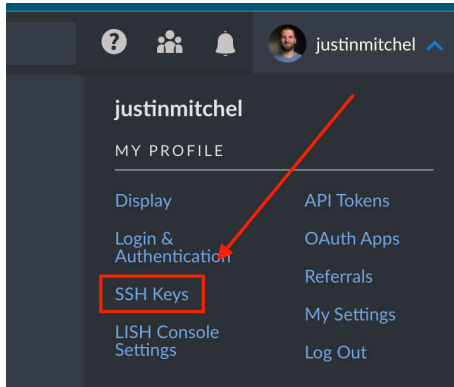


Figure 4.7 Akamai Linode navigation bar: click SSH Keys.

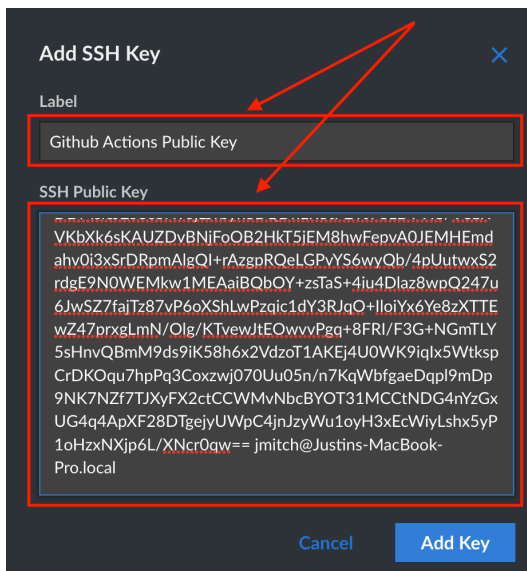


Figure 4.8 Akamai Linode: paste new SSH key.

A more in-depth guide on SSH keys can be found in appendix D. With our SSH keys installed, let's provision a new virtual machine on Akamai Connected Cloud.

4.1.4 Creating a new virtual machine on Akamai Linode

Now that we have our SSH keys in place on Akamai Connected Cloud and GitHub Actions secrets, we will provision a new virtual machine instance. As you may recall, we did this exact step in section 3.1.1, so we'll keep this section brief.

Navigate to Akamai Connected Cloud Linodes (<https://cloud.linode.com/linodes>) and click Create Linode. Use the following settings:

- Distribution—Ubuntu 22.04 LTS (or the latest LTS).
- Region—Dallas, TX (or the region nearest you).
- Plan—Shared CPU > Nanode 1 GB (or larger).
- Label—rk8s-github-actions.
- Tags—<optional>.
- Root Password—Set a good one.
- SSH Keys—Select your newly created SSH Key (e.g., *GitHub Actions Public Key*) and any other user keys you may need, as seen in figure 4.9.
- All other settings—<optional>.

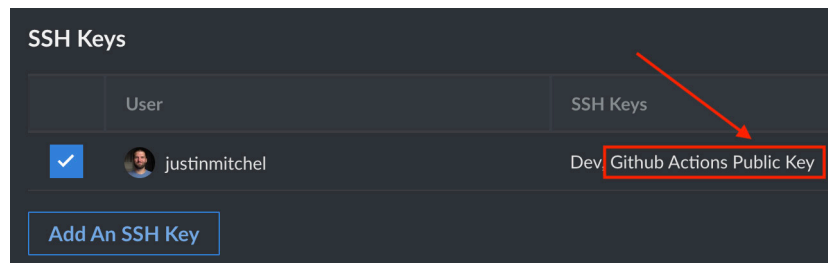


Figure 4.9 Akamai Linode: create VM GitHub Actions with SSH keys.

After you configure these settings, click Create Linode and wait for the IP address to be assigned, as seen in figure 4.10. Once the IP address is assigned, we will move on to the next section, where we use GitHub Actions to install NGINX on this virtual machine.

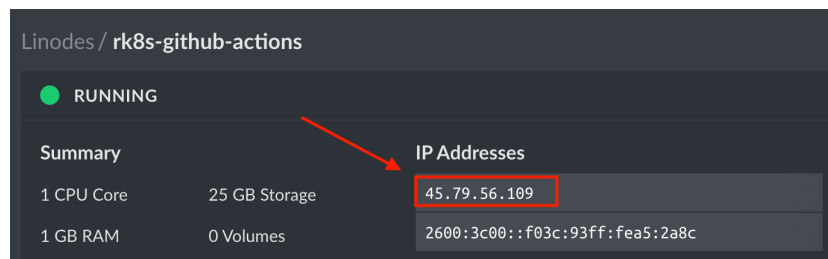


Figure 4.10 Akamai Linode: IP address assigned to GitHub Actions

Figure 4.10 shows the IP address of the newly provisioned virtual machine as `45.79.56.109`. This IP address will be different for you, so be sure to use the IP address assigned to your virtual machine.

Back in your GitHub Repo's Actions secrets, you need to add this IP address just like we did the SSH private key in section 4.1.3. Use the secret name of `AKAMAI_INSTANCE_IP_ADDRESS` and the secret value with your IP address as seen in figure 4.11.

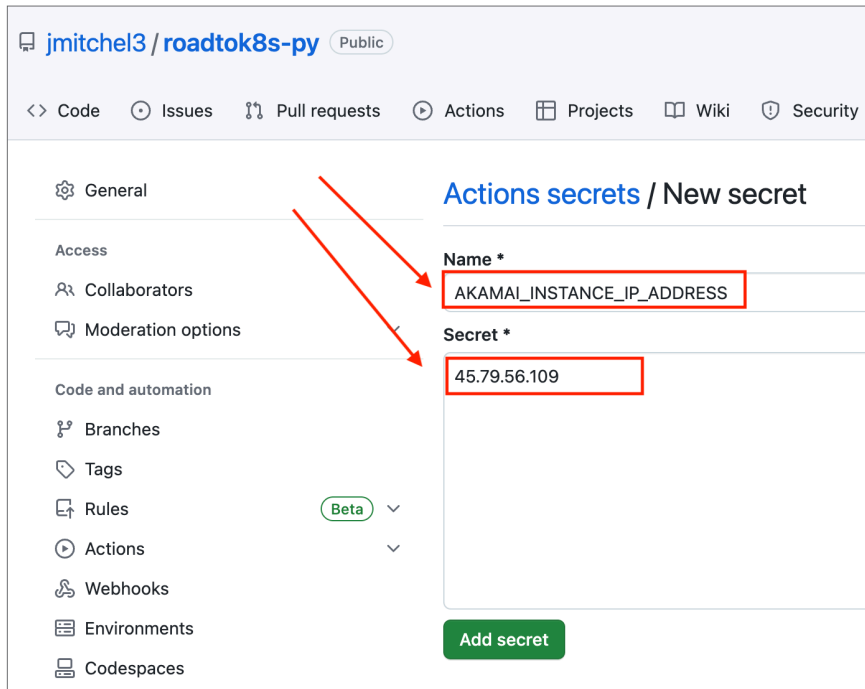


Figure 4.11 GitHub Actions: add IP address secret.

With all of the conditions set correctly, we are now going to create a GitHub Actions workflow that will install NGINX on our newly created virtual machine. The workflow and process are as simple as it gets, but it's an important next step to using automation pipelines.

4.1.5 *Installing NGINX on a remote server with a GitHub Actions workflow*

The standard way to install NGINX on Ubuntu systems is as simple as they come: first we use `sudo apt update` and then `sudo apt install nginx` (or `apt-get install nginx`). These commands should look very familiar to you, because we used these in section 3.2. This simplicity is a great way to test our progress in adopting new methods of deployment—for example, using GitHub Actions.

Knowing that these are the two commands, we can use them with the built-in `ssh` command in the format of `ssh <user>@<ip-address> <command>`. For example, if we

wanted to run `sudo apt update` and `sudo apt install nginx -y` on our newly provisioned virtual machine, we would run the following commands:

```
ssh root@my-ip-address sudo apt get update:
ssh root@my-ip-address sudo apt get install nginx -y
```

Remember, the `-y` will automatically approve any prompts during this installation, thus making it fully automated, assuming the `nginx` package exists.

With these two commands in mind, our new workflow will use two of our GitHub Actions secrets:

- `SSH_PRIVATE_KEY`—This is the private key we created in section 4.1.2.
- `AKAMAI_INSTANCE_IP_ADDRESS`—This is the IP address of the virtual machine we created in section 4.1.3.

We *must never* expose secrets in GitHub Actions or Git repositories. Secrets are meant to be *secret* and should be treated as such. Storing secrets in GitHub Actions is a great way to keep them secure, out of your codebase, and out of the hands of others.

Using stored secrets within a GitHub Actions workflow requires special syntax that looks like this: `${{ secrets.SECRET_NAME }}` so we'll use the following:

- `${{ secrets.SSH_PRIVATE_KEY }}`—for our SSH private key
- `${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}`—for our Akamai instance IP address

Passwordless SSH sessions require at least the following conditions:

- *The private key*—Available locally (e.g., `cat ~/.ssh/id_rsa`) and executable with `chmod 600 ~/.ssh/id_rsa`
- *The public key*—Installed remotely (e.g., in `~/.ssh/authorized_keys`)
- *An IP address (or hostname)*—Accessible via the internet (or simply to the machine attempting to connect)

We have the conditions set for passwordless SSH sessions, but we still need to install the private key during the execution of a GitHub Actions workflow. This means we will inject this key during the workflow so future steps can use this configuration. Luckily for us, this is a straightforward process, as we can see in listing 4.4. Create a new workflow file at the path `.github/workflows/install-nginx.yaml` with the contents of the following listing.

Listing 4.4 Installing NGINX with GitHub Actions

```
name: Install NGINX
on:
  workflow_dispatch:
  push:
```

```

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Configure the SSH Private Key Secret
        run: |
          mkdir -p ~/.ssh/
          echo "${{ secrets.SSH_PRIVATE_KEY }}" > ~/.ssh/id_rsa
          chmod 600 ~/.ssh/id_rsa
      - name: Set Strict Host Key Checking
        run: echo "StrictHostKeyChecking=no" > ~/.ssh/config
      - name: Install NGINX
        run: |
          export MY_HOST="${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}"
          ssh root@$MY_HOST sudo apt update
          ssh root@$MY_HOST sudo apt install nginx -y

```

As we can see, this workflow introduced two new steps that we did not have in our *Hello World* workflow, while everything else remained virtually untouched. Let's review the new steps:

- 1 *Configure the SSH private key secret*—Once again, we see that we use `run:` to execute a command. In this case, we use a multiline command by leveraging the pipe (`|`) to enable multiple commands within this single step. We *could* break these commands into separate steps for more granular clarity on each step (e.g., if one of the commands fails, often having multiple steps can help).
- 2 *Set strict host key checking*—Typically, when you connect via SSH on your local machine, you are prompted to verify the authenticity of the host key. This step skips that verification process because GitHub Actions workflows do not allow for user input. You could also consider using

```
echo "${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}" $(ssh-keyscan -H
${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}) " > ~/.ssh/known_hosts
```

to add the host key to the known host file instead of

```
echo "StrictHostKeyChecking=no" > ~/.ssh/config
```

- 3 *Install NGINX*—This step is very similar to the previous step, except we do not need to configure the SSH private key secret again because the entire `build` job has access to the results of previous steps. We simply run the two commands to install NGINX on our remote server.

Now you can open your web browser to your IP address and see the NGINX Hello World page, as we have seen before. Seeing this page might not be too exciting because we've seen it before, and there's not much going on here, but the fact that we did this with GitHub Actions is a big deal. We can now use GitHub Actions to install software on remote servers, and we can do it in a repeatable and automated fashion. This is a big step toward a fully automated CI/CD pipeline.

At this point, we could dive into the wonderful world of `ssh` and `scp` (a command to copy files) to install and configure our remote host, but instead, we're going to implement a more robust solution, Ansible, that will allow us to install and configure our remote host. From a developer's perspective, Ansible and GitHub Actions look a lot alike because they both use YAML-formatted files, but Ansible is designed to automate the configuration of remote hosts.

4.2 Virtual machine automation with Ansible

Ansible is an enjoyable way to install and configure machines, and in this section, I will show you exactly why. When we consider the command `sudo apt install nginx -y`, a few questions come to mind:

- Is NGINX installed?
- Is NGINX running?
- Do we need to add or update our NGINX configuration?
- Will we need to restart or reload NGINX after we configure it, if we do need to configure it?
- Do we need to perform this *same exact process* on another machine? How about 10 others? 1,000 others?

Since this command, `sudo apt install nginx -y`, raises several really good questions, we must consider what they are all about: *desired state*. Desired state is all about ensuring any given machine is configured exactly as needed so our applications will run as intended. I picked *NGINX* because it's an incredibly versatile application that has very little overhead to get working well. NGINX is also a great stand-in for when we need to install and configure more complex applications like our Python and Node.js applications. NGINX also gives us features that we can use on both of our applications, such as load balancing and path-based routing.

Ansible, like other infrastructure-as-code (IaC) software, is a tool that helps us achieve a desired state on our machines regardless of the *current state* of those machines so we can reach a specific outcome through YAML-based files called *Ansible Playbooks*. Ansible is an example of *declarative* programming. We will use Ansible to do the following:

- Install OS-level and project-specific dependencies for Python, Node.js, NGINX, and Supervisor.
- Configure Supervisor and NGINX.
- Start, restart, and reload Supervisor and NGINX.

Ansible works through a secure shell connection (SSH), so it remains critical that we have the proper SSH keys on the machine to run Ansible and the machine or machines we aim to configure.

Ansible is a powerful tool with many options, so I recommend keeping the Official Documentation (<https://docs.ansible.com/ansible/latest/>) handy to review all kinds

of configuration options. Let's create our first Ansible Playbook along with a GitHub Actions workflow to run it.

4.2.1 **GitHub Actions workflow for Ansible**

Ansible is a Python-based tool that we can install on nearly any machine, and in our case, we will continue using GitHub Actions workflows as our configuration machine and CI/CD pipeline. Using Ansible's full capabilities is well beyond the scope of this book, but we will use the minimal amount to achieve the results we're after.

Before we create our first Ansible Playbook, we will create a GitHub Actions workflow that will establish all the pieces we need to *run* our Ansible Playbook. This workflow will do the following:

- 1 Set up Python 3.
- 2 Install Ansible.
- 3 Implement the private SSH key.
- 4 Create an Ansible inventory file for our remote host.
- 5 Create a Ansible default configuration file.
- 6 Run any Ansible Playbook.

Before we look at the GitHub Actions Workflow, let's review the new terms:

- *Inventory file*—Ansible has the flexibility to configure a lot of remote hosts via an IP address or even a domain name. This file is how we can define and group the remote hosts as we see fit. In our case, we'll use one remote host within one group.
- *Default configuration file*—Configuring Ansible to use the custom inventory file we create, along with a few SSH-related options, is all we need to get started. We'll use the default configuration file to do this.
- *Ansible Playbook*—Playbooks are the primary feature of Ansible we'll be using. These are YAML-formatted files that define the desired state of a remote host or group of remote hosts. For larger projects, playbooks can get rather complex.

In listing 4.5, we see the workflow file that will set up Ansible and our remote host. Create a new workflow file at the path `.github/workflows/run-ansible.yaml` with the contents of the following listing.

Listing 4.5 Running Ansible with GitHub Actions

```
name: Run Ansible
on:
  workflow_dispatch:

jobs:
  run-playbooks:
```

```

runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v3
- name: Setup Python 3
  uses: actions/setup-python@v4
  with:
    python-version: "3.8"
- name: Upgrade Pip & Install Ansible
  run: |
    python -m pip install --upgrade pip
    python -m pip install ansible
- name: Implement the Private SSH Key
  run: |
    mkdir -p ~/.ssh/
    echo "${{ secrets.SSH_PRIVATE_KEY }}" > ~/.ssh/id_rsa
    chmod 600 ~/.ssh/id_rsa
- name: Ansible Inventory File for Remote host
  run: |
    mkdir -p ./devops/ansible/
    export INVENTORY_FILE=./devops/ansible/inventory.ini
    echo "[my_host_group]" > $INVENTORY_FILE
    echo "${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}" >> $INVENTORY_FILE
- name: Ansible Default Configuration File
  run: |
    mkdir -p ./devops/ansible/
    cat <<EOF > ./devops/ansible/ansible.cfg
    [defaults]
    ansible_python_interpreter = '/usr/bin/python3'
    ansible_ssh_private_key_file = ~/.ssh/id_rsa
    remote_user = root
    inventory = ./inventory.ini
    host_key_checking = False
    EOF
- name: Ping Ansible Hosts
  working-directory: ./devops/ansible/
  run: |
    ansible all -m ping
- name: Run Ansible Playbooks
  working-directory: ./devops/ansible/
  run: |
    ansible-playbook install-nginx.yaml
- name: Deploy Python via Ansible
  working-directory: ./devops/ansible/
  run: |
    ansible-playbook deploy-python.yaml

```

The two key files created in this workflow are `inventory.ini` and `ansible.cfg`. This workflow is an end-to-end example of what needs to be done to run an Ansible workflow. The command `ansible all -m ping` simply ensures that our previous configuration was done correctly and in the correct location. We see the declaration `working-directory` for each of the `ansible` commands (`ansible` and `ansible-playbook`) simply because `ansible.cfg` is located in and references the `inventory.ini` file in the `working-directory`.

After you commit this workflow to Git, push it to GitHub, and run it, you will see the result in figure 4.12.



```

1 ▶ Run ansible all -m ping
11 *** | SUCCESS => {
12   "ansible_facts": {
13     "discovered_interpreter_python": "/usr/bin/python3"
14   },
15   "changed": false,
16   "ping": "pong"
17 }

1 ▶ Run ansible-playbook install-nginx.yaml
11 ERROR! the playbook: install-nginx.yaml could not be found
12 Error: Process completed with exit code 1.

```

Figure 4.12 GitHub Actions Ansible setup result

Ansible can verify that it's connecting to our remote host and that it fails to run the `install-nginx.yaml` workflow (because it hasn't been created yet). If you see the same result, then you're ready to create your first Ansible Playbook. If you see anything related to connecting to *localhost*, there's a good chance that your `inventory.ini` file was incorrectly created or the IP address is no longer set correctly in your GitHub Actions secrets. If you see anything related to *permission denied* or *ssh key errors*, there's a good chance that your SSH private key is not set up correctly.

Assuming you got the result shown in figure 4.12, we're now ready to create our first Ansible Playbook. Get ready because this is where the magic happens.

4.2.2 **Creating your first Ansible Playbook**

In my experience, basic Ansible Playbooks are much easier to understand than GitHub Actions workflows, but advanced Ansible Playbooks are much harder to understand. Luckily for us, we'll stick with basic Ansible Playbooks for the remainder of this book. In each playbook, we'll declare the following:

- **name**—The *name* of the playbook.
- **hosts**—The *hosts* and the *host group* that this playbook will run on. You can use `all`, which means *all hosts*, or you can use a specific group such as `[my_host_group]`, which we defined in `inventory.ini` in listing 4.5. Using `all` is what we can use for our first playbook.
- **become**—a special keyword that allows us to run commands as the *root* user. Using a root user is not always recommended for production environments, but the intricacies of user permissions are beyond the scope of this book.
- **tasks**—a list of configurations we want Ansible to run. Each task runs after the previous task.

Listing 4.6 shows the contents of `install-nginx.yaml`, which we will create at the path `devops/ansible/install-nginx.yaml`.

Listing 4.6 Installing NGINX with Ansible

```

- name: Install NGINX
  hosts: all
  become: true
  tasks:
    - name: Install NGINX
      apt:
        name: nginx
        state: present
        update_cache: yes

```

Are you floored by how simple this is? Probably not. After all, it's a lot more lines than just `sudo apt update && sudo apt install nginx -y`. The major difference is that we can run this same workflow on thousands of hosts if needed, with the same ease as running on one host (assuming the correct SSH keys are installed, of course). We can make this even more robust by ensuring that the NGINX service is actually running on this host by appending a new task to this playbook, as shown in the following listing.

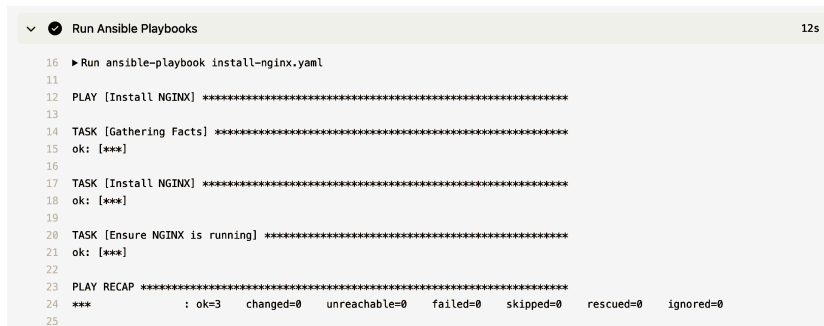
Listing 4.7 Ensuring NGINX is running with Ansible

```

- name: Install NGINX
  hosts: all
  become: true
  tasks:
    - name: Install NGINX
      apt:
        name: nginx
        state: present
        update_cache: yes
    - name: Ensure NGINX is running
      service:
        name: nginx
        state: started
        enabled: yes

```

With this file created, let's commit it to Git, push it, and then run our GitHub Actions Ansible workflow. You should see the output shown in figure 4.13.



The screenshot shows the output of a GitHub Actions workflow named 'Run Ansible Playbooks'. The output is displayed in a terminal window with line numbers 16 through 25. The workflow successfully runs the Ansible playbook 'install-nginx.yaml'. The output shows the following steps:

- PLAY [Install NGINX]**: This section contains three tasks:
 - TASK [Gathering Facts]**: Completed successfully with 'ok: [***]'.
 - TASK [Install NGINX]**: Completed successfully with 'ok: [***]'.
 - TASK [Ensure NGINX is running]**: Completed successfully with 'ok: [***]'.
- PLAY RECAP**: Shows the final status of the run: '*** : ok=3 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0'.

The workflow completed successfully, as indicated by the green checkmark icon and the '12s' duration.

Figure 4.13 GitHub Actions Ansible install NGINX

With Ansible, we can see the status of each task being run in the order it was declared. If you had more than one host in your inventory file, you would see the same output for each host. This is a great way to see the status of each task and each host. Once again, we see the *desired state* (e.g., the declared task), along with the *current state*, and the *updated state* if it changed. This is a great way to see what Ansible is doing and why it's doing it. Of course, adding more remote hosts is as easy as the following:

- 1 Add each host as a secret in GitHub Actions Secrets (e.g., `AKAMAI_INSTANCE_IP_ADDRESS_1`, `AKAMAI_INSTANCE_IP_ADDRESS_2`, etc.).
- 2 Add each host to the `inventory.ini` file in the workflow (e.g., `echo "${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS_1 }}" >> inventory.ini` and `echo "${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS_2 }}" >> inventory.ini`).
- 3 Run the GitHub Actions workflow.

With this new Ansible Playbook, we can now install NGINX on any number of remote hosts with a single command. This is a great step toward a fully automated CI/CD pipeline. Now we need to take the next step toward automating the deployment of our Python and Node.js applications.

4.2.3 *Ansible for our Python app*

In chapter 3, we deployed a Python application and used a number of Git hooks to perform the configuration. While this method works and is valid, it's prone to errors and requires us to self-host our Git repo, which is not ideal (for a large number of reasons).

Before we start creating the playbook, let's recap what we need to do for our Python application to run on nearly any remote host:

- Install Python 3 and related system-wide dependencies (such as `python3-dev`, `python3-venv`, `build-essential`, etc.).
- Create a Python 3.8+ virtual environment.
- Copy or update the source code.
- Install or update the requirements (via Python's `requirements.txt`).
- Install and configure Supervisor for the Python app.
- Install and configure NGINX for the Python app.

The source code and the `requirements.txt` file are the items that will likely change the most, while the configurations for Supervisor and NGINX are likely to change very little. We'll start by adding a purpose-designed configuration for each tool by adding the following files to our local repo:

- `conf/nginx.conf`
- `conf/supervisor.conf`

The configuration for Supervisor at `conf/supervisor.conf` comes directly from listing 3.27. The idea is to use a specific command to run our Python application (e.g.,

gunicorn) along with a working directory and a location for the log output, which is all encapsulated in the following listing.

Listing 4.8 Supervisor configuration for Python and Ansible

```
[program:roadtok8s-py]
directory=/opt/projects/roadtok8s/py/src
command=/opt/venv/bin/gunicorn
    --worker-class uvicorn.workers.UvicornWorker main:app
    --bind "0.0.0.0:8888" --pid /var/run/roadtok8s-py.pid
autostart=true
autorestart=true
startretries=3
stderr_logfile=/var/log/supervisor/roadtok8s/py/stderr.log
stdout_logfile=/var/log/supervisor/roadtok8s/py/stdout.log
```

NOTE For those familiar with writing multiline bash commands, you will notice that `command=` does not have the new line escape pattern (`\n`) on each line. This is done on purpose because of how Supervisor parses this file.

The next piece is to implement the NGINX configuration at `conf/nginx.conf` with a few slight modifications to what you see in listing 4.9.

Listing 4.9 NGINX configuration for Python and Ansible

```
server {
    listen 80;
    server_name _;
    location / {
        proxy_pass http://127.0.0.1:8888;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Let's create our Ansible file piece by piece to understand what's going on. First, we'll create a new Ansible Playbook at the path `devops/ansible/deploy-python.yaml` with the contents of listing 4.10.

Listing 4.10 Installing Python with Ansible Playbook, part 1

```
- name: Deploy Python app
  hosts: all
  become: yes
```

Now let's move on to the first task of updating the system's *package cache*. We do this first so that future tasks can use this if needed. Listing 4.11 shows the first task. Define package cache and explain briefly under what conditions future tasks will need to use it.

Listing 4.11 Installing Python with Ansible Playbook, part 2

```
tasks:
  - name: Update and upgrade system
    apt:
      upgrade: yes
      update_cache: yes
      cache_valid_time: 3600
```

This will update the system's package cache for about an hour. In the next listing, we'll install the specific apt packages our system needs to run Python, Supervisor, and NGINX.

Listing 4.12 Installing Python with Ansible Playbook, part 3

```
- name: Install dependencies
  apt:
    pkg:
      - python3-pip
      - python3-dev
      - python3-venv
      - rsync
      - nginx
      - supervisor
    state: present
```

The `apt` declaration is built-in to Ansible and allows us to have the `state: present` line as an option on this task for any given package we list. We have other options for `state` (e.g., `latest`, `absent`). What's powerful about this `state: present` line is how repeatable it is, assuming `apt install` is your OS-level install package. This is a lot like what `requirements.txt` enables for us within our Python project but for our entire system.

Since `python3-venv` is a core component of the installations done in listing 4.12, we can now create our Python 3.8+ virtual environment. Listing 4.13 shows the task to create the virtual environment.

Listing 4.13 Installing Python with Ansible Playbook, part 4

```
- name: Create virtual environment
  ansible.builtin.command:
    cmd: python3 -m venv /opt/venv
    creates: /opt/venv
```

The built-in Ansible `command` module (see *command module* in the Ansible documentation for more details) runs a command without shell variables (like bash variables `$HOME`, `$PWD`, `$USER`). If you are familiar with these bash variables and need them, you can use `ansible.builtin.shell` instead. In this task, it will run the command `python3 -m venv /opt/venv` only if the directory `/opt/venv` does not exist due to the

`creates:` declaration with the folder the virtual environment creates. This is a great way to ensure that this task is only run once to guard against running it multiple times, which can cause problems with other commands.

Now we want to create the directories for both the source code (e.g., the Python project) and the log output folder(s) as specified in Supervisor. If you need to run a task multiple times for multiple directories, you can use the `with_items` declaration, as shown in the following listing.

Listing 4.14 Installing Python with Ansible Playbook, part 5

```
- name: Set up application directory
  file:
    path: "{{ item }}"
    state: directory
    owner: "{{ ansible_user }}"
    group: "{{ ansible_user }}"
    mode: '0755'
  with_items:
    - /opt/projects/roadtok8s/py/src/
    - /var/log/supervisor/roadtok8s/py/
```

As we see here, `with_items` takes a list of items and iterates over the remainder of the task. In this case, the task is to create a file or directory. This task introduces us to Jinja2-like syntax with the variable `{{ item }}`, which corresponds directly to the list of items in `with_items`, and the `{{ ansible_user }}` variable, which is the `root` user, as provided directly by Ansible because of the `become: yes` declaration at the top of the playbook. The rest of the `file` declaration is just to ensure that each directory is created with the correct permissions and ownership so that our user (and other applications) can properly use these directories.

Now we have a choice to make: How do we get our code to our remote machine(s)? On the one hand, we could use Git on the remote machine(s) to clone or pull the code, or we could synchronize the code from the local machine (e.g., the GitHub Actions workflow). We will opt to synchronize the code for the following reasons:

- Our GitHub Actions workflow already has a Git checkout copy of the most current code.
- Ansible has a clean built-in way to synchronize files and directories (without needing to copy each one each time).
- Within GitHub Actions workflows, our Ansible `synchronize` command is less prone to accidentally synchronizing files and folders that should not be synchronized (e.g., any file that is not yet managed by Git and is not yet in `.gitignore`).
- GitHub Actions workflows can be run on private code repositories. Attempting to clone or pull a private repo on a remote machine requires additional configuration and is not as clean as using Ansible's `synchronize` command.

Whenever I need to copy a number of files to a remote host with Ansible, I tend to use the built-in `synchronize` module (see *synchronize module* in the Ansible documentation for more details) because it handles recursively copying directories very well. The following listing shows the task to synchronize the code from the local machine to the remote machine.

Listing 4.15 Installing Python with Ansible Playbook, part 6

```
- name: Copy FastAPI app to remote server
  synchronize:
    src: '{{ playbook_dir }}/../../src/'
    dest: /opt/projects/roadtok8s/py/src/
    recursive: yes
    delete: yes
```

We're introduced to another built-in Ansible variable called `{{ playbook_dir }}`, which is the location where the playbook is being executed. In this case, the `playbook_dir` variable references where this playbook is located, which is in `devops/ansible/`. This means we need to synchronize the `src` folder from two levels up, which is where `../../src/` comes in. `dest` is the location on the remote machine where we want to synchronize the files. In this case, we want to synchronize the files to `/opt/projects/roadtok8s/py/src/`, which is the same location we created in listing 4.16.

We use `recursive: yes` and `delete: yes` to ensure that every file in `src/` is the same from GitHub to our remote host(s), thus keeping the source code in sync. Storing our source code in `src/` is a common convention for Python projects and is a great way to keep our source code separate from our configuration files and thus make it easier to synchronize with Ansible.

Now we can move on and install all of the Python project packages from `requirements.txt`, and if the `synchronize` task worked correctly, that file will be located at `/opt/projects/roadtok8s/py/src/requirements.txt`. We can use this file along with the `pip` executable located at `/opt/venv/bin/pip` to install our Python packages to our Python virtual environment. Listing 4.16 shows the task to install the Python packages.

Listing 4.16 Installing Python with Ansible Playbook, part 7

```
- name: Install Python packages in virtual environment
  ansible.builtin.pip:
    requirements: /opt/projects/roadtok8s/py/src/requirements.txt
    executable: /opt/venv/bin/pip
```

Python *pip* will not install or update a package that is already installed, so it's perfectly reasonable to attempt to run this task on every Ansible run. This is a great way to ensure that the Python packages are always up to date. There are more advanced configurations that can be done with *pip*, but this is the approach we'll take.

At this point, our remote host(s) would have our Python project installed and ready to run. All we have to do now is copy our NGINX and Supervisor configuration. When

they are copied, we'll also notify an Ansible feature called a *handler* to run a task based on these configuration changes. Ansible handlers are basically callback blocks we can trigger from anywhere in our playbooks using the `notify:` argument. Listing 4.17 shows the task to copy these configurations.

Listing 4.17 Installing Python with Ansible Playbook, part 8

```
- name: Configure gunicorn and uvicorn with supervisor
  copy:
    src: '{{ playbook_dir }}/../../conf/supervisor.conf'
    dest: /etc/supervisor/conf.d/roadtok8s-py.conf
    notify: reload supervisor
- name: Configure nginx
  copy:
    src: '{{ playbook_dir }}/../../conf/nginx.conf'
    dest: /etc/nginx/sites-available/roadtok8s-py
    notify: restart nginx
```

When I am copying a single file, I tend to use the built-in `copy` module (see *copy module* in the Ansible documentation for more details) as seen in listing 4.17. This module is very similar to the `file` module we used in listing 4.16, but it's more specific to copying files. If the supervisor configuration file changes, use the `notify` declaration to trigger the `reload supervisor` handler. If the `nginx` configuration changes, we trigger the `restart nginx` handler. Each handler is defined outside of the `tasks:` block and is run after all tasks have completed. Almost any task or handler can trigger a handler to execute. In this case, it makes sense that if our NGINX or Supervisor configuration changes, we should reload or restart each respective service.

With our new NGINX configuration added, we will remove the default NGINX configuration and link our new configuration to the `sites-enabled` directory. The following listing shows the task to do this.

Listing 4.18 Installing Python with Ansible Playbook, part 9

```
- name: Enable nginx site
  command: ln -s /etc/nginx/sites-available/roadtok8s-py /etc/nginx/sites-enabled
  args:
    creates: /etc/nginx/sites-enabled/roadtok8s-py
- name: Remove default nginx site
  file:
    path: '{{ item }}'
    state: absent
  notify: restart nginx
  with_items:
    - /etc/nginx/sites-enabled/default
    - /etc/nginx/sites-available/default
```

Removing the default NGINX configuration is optional, but it's often a good idea, as it ensures that only your NGINX configuration is being used. We use the `file` module

to remove the default configuration and then use the `command` module again to create a symbolic link from our new configuration to the `sites-enabled` directory. This is a common way to configure NGINX and is a great way to ensure that our NGINX configuration is always up to date.

Now it's time to implement the handler for each service. Notifying handlers typically occurs when a task with `notify:` runs successfully *and* the handler referenced exists. In our case, we used `notify: restart nginx` and `notify: reload supervisor`. We must remember that the *handler name* must match the `notify:` declaration exactly (e.g., `restart nginx` and `reload supervisor`). Listing 4.19 shows the handler for reloading Supervisor and restarting NGINX.

Listing 4.19 Installing Python with Ansible Playbook, part 10

```
handlers:
  - name: reload supervisor
    command: "{{ item }}"
    with_items:
      - supervisorctl reread
      - supervisorctl update
      - supervisorctl restart roadtok8s-py
  notify: restart nginx
  - name: restart nginx
    systemd:
      name: nginx
      state: restarted
```

Handlers are callback functions that are only run if they are notified. Handlers are only notified if a block (task or another handler) successfully executes. To notify a handler, we use the `notify: <handler name>` declaration, assuming the `<handler name>` is defined and exists in the playbook.

In our case, the handler *reload supervisor* has a `notify: restart nginx` definition. This means that if the `reload supervisor` handler is called and is successful then the `restart nginx` handler will be triggered. If the `restart nginx` handler had a `notify: reload supervisor`, we would see an infinite loop between these handlers. This infinite loop should be avoided, but it is something to be aware of.

With the playbook nearly complete, let's commit it to Git and push it to GitHub. Once you do, you should have the following files related to running Ansible:

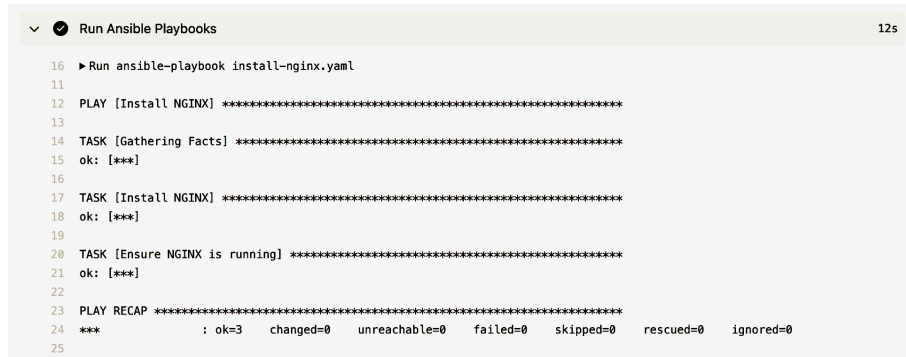
- `devops/ansible/install-nginx.yaml`
- `devops/ansible/deploy-python.yaml`
- `.github/workflows/run-ansible.yaml`

As you may recall, `run-ansible.yaml` has a step that calls the `install-nginx.yaml` file with `ansible-playbook install-nginx.yaml`. We must update this line to read `ansible-playbook deploy-python.yaml` instead. The following listing shows the updated step in the workflow file.

Listing 4.20 Deploying the Python app with Ansible with GitHub Actions

```
- name: Deploy Python via Ansible
  working-directory: ./devops/ansible/
  run: |
    ansible-playbook deploy-python.yaml
```

Before we commit this playbook and updated workflow to our GitHub Repo, be sure to compare yours with the full reference playbook at my Python project's official repo at <https://github.com/jmitchel3/roadtok8s-py>. With the full playbook and the updated GitHub Actions workflow, it is time to run the workflow and see what happens. When you run the workflow, you should see the output in figure 4.14.



The screenshot shows the output of a GitHub Actions workflow named 'Run Ansible Playbooks'. The workflow is completed successfully, as indicated by a green checkmark and the text 'Run Ansible Playbooks' in the top bar. The output is displayed in a scrollable area with line numbers 16 through 25. The output shows the execution of the 'install-nginx.yaml' playbook. The first task is 'Gathering Facts', which is successful. The second task is 'Install NGINX', which is also successful. The third task is 'Ensure NGINX is running', which is successful. The final output is a 'PLAY RECAP' showing that 3 tasks were successful, and no tasks were changed, unreachable, failed, skipped, rescued, or ignored.

```
16 ▶ Run ansible-playbook install-nginx.yaml
17
18 PLAY [Install NGINX] *****
19
20 TASK [Gathering Facts] *****
21 ok: [***]
22
23 TASK [Install NGINX] *****
24 ok: [***]
25
26 TASK [Ensure NGINX is running] *****
27 ok: [***]
28
29 PLAY RECAP *****
30 *** : ok=3    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 4.14 GitHub Actions Ansible deploy Python result

If you ran into errors with this workflow, you may have to use a manual SSH session to diagnose all the problems that may have occurred, just like we did in chapter 3. If you did not run into any errors, you should be able to open your web browser to your IP address and see the Python application running. If you see the Python application running, you have successfully deployed your Python application with Ansible and GitHub Actions. Congratulations!

If you were successful, I recommend changing your Python source code and running it all over again. Ensure that the desired outcome(s) are happening correctly. The next step is to implement this Ansible and GitHub Actions automation on the Node.js application.

4.2.4 Ansible for our Node.js app

Deploying a Node.js application with Ansible and GitHub Actions is almost identical to deploying the Python application, with only a few exceptions. We are going to create a new virtual machine for this process, although we could deploy this application to the same virtual machine as the Python project, as we saw in chapter 3. This section will be far more condensed than the previous sections because we can reuse a lot of our code and configurations.

Let's recap what we need to do in order for our Node.js application to run on GitHub Actions correctly:

- 1 *Create new SSH keys.* Having new keys will help ensure additional security for each isolated web application.
 - Add the new SSH private key as a GitHub Actions secret on our Node.js repo.
 - Install the new SSH public key on Akamai Linode.
 - Provision a new virtual machine and add the new IP address as a GitHub Actions secret.
- 2 Copy the NGINX and Supervisor configuration files for Node.js from chapter 3.
- 3 Create a new Ansible Playbook for our Node.js application.
- 4 Copy and modify the GitHub Actions workflow for running Ansible, created in section 4.2.3.

You should be able to do steps 1-2 with ease now, so I will leave that to you. Now let's go ahead and create our Ansible Playbook for our Node.js application.

CREATING A NEW ANSIBLE PLAYBOOK FOR OUR NODE.JS APPLICATION

We already have most of the foundation necessary to create a Node.js-based Ansible Playbook for our project. The main difference between using Ansible for installing Node and what we did in chapter 3 is that we are going to update the `apt` package directly instead of using the Node Version Manager.

In your Node.js application at `~/dev/roadtok8s/js`, create a folder `devops` and add the file `deploy.js.yaml`, and declare the first few steps as outlined in listing 4.21. Since `nvm` and Ansible do not play well together, we added a new block called `vars` (for variables) that we can reuse throughout this playbook. In this case, we set the variable `nodejs_v` to the value `18` to be used to install Node.js v18, as shown in the following listing.

Listing 4.21 Installing Node.js with Ansible Playbook, part 1

```
---
- name: Deploy Node.js app
  hosts: all
  become: yes
  vars:
    nodejs_v: 18
  tasks:
    - name: Update and upgrade system
      apt:
        upgrade: yes
        update_cache: yes
        cache_valid_time: 3600
    - name: Install dependencies
      apt:
        pkg:
          - curl
          - rsync
          - nginx
```

```
- supervisor
  state: present
  tags: install
```

Once again, we start will updating `apt` and various system requirements to set the foundation for installing Node.js.

Unfortunately, we cannot just install Node.js with `apt install nodejs`, so we need to add a new key and apt repository with Ansible by using the blocks that add a new `apt_key` and `apt_repository`, respectively. Once this is done, we can install Node.js using the `apt` block directly, as seen in the following listing.

Listing 4.22 Installing Node.js with Ansible Playbook, part 2

```
---
- name: Deploy Node.js app
  hosts: all
  become: yes
  vars:
    nodejs_v: 18
  tasks:
    - name: Update and upgrade system
      apt:
        upgrade: yes
        update_cache: yes
        cache_valid_time: 3600
    - name: Install dependencies
      apt:
        pkg:
          - curl
          - rsync
          - nginx
          - supervisor
        state: present
        tags: install
```

Once this is complete, we'll use the `npm` block to update npm globally, as seen in the following listing.

Listing 4.23 Installing Node.js with Ansible Playbook, part 3

```
- name: Update npm globally
  npm:
    global: yes
    name: "{{ item }}"
    state: latest
  with_items:
    - npm
```

The `npm` block is built in to Ansible, but our host machine must have it installed; otherwise, it will not succeed.

Before we can proceed further, we need to update our `supervisor.conf` to account for the change in how we start our Node.js application. You can simply change the command from `/root/.nvm/versions/node/v18.16.0/bin/node main.js` to `command=node main.js`. With the `supervisor.conf` file updated, we can start synchronizing the code to the host machine, as seen in the following listing.

Listing 4.24 Installing Node.js with Ansible Playbook, part 4

```
- name: Set up application directory
  file:
    path: "{{ item }}"
    state: directory
    mode: '0755'
  with_items:
    - /opt/projects/roadtok8s/js/src/
    - /var/log/supervisor/roadtok8s/js/
- name: Copy Node.js app to remote server
  synchronize:
    src: '{{ playbook_dir }}/../../src/'
    dest: /opt/projects/roadtok8s/js/src/
    recursive: yes
    delete: yes
- name: Copy Package.json to remote server
  synchronize:
    src: '{{ playbook_dir }}/../../package.json'
    dest: /opt/projects/roadtok8s/js/package.json
    recursive: yes
    delete: yes
```

This process is nearly identical with what we did in listing 4.14 and 4.15, just modified slightly to account for the Node.js application.

With the code synchronized, we can now install the Node.js application dependencies with the `npm` block, as seen in the following listing.

Listing 4.25 Installing Node.js with Ansible Playbook, part 5

```
- name: Install Node.js packages
  shell: |
    npm install
  args:
    executable: /bin/bash
    chdir: "/opt/projects/roadtok8s/js/"
  notify: reload supervisor
```

The remainder of this file is the same as the one for the Python application before it, as you can see in the following listing.

Listing 4.26 Installing Node.js with Ansible Playbook, part 6

```
- name: Configure Node.js with supervisor
  copy:
```



```

    src: ../../conf/supervisor.conf
    dest: /etc/supervisor/conf.d/roadtok8s-js.conf
  notify: reload supervisor
- name: Configure nginx
  copy:
    src: ../../conf/nginx.conf
    dest: /etc/nginx/sites-available/roadtok8s-js
  notify: restart nginx
- name: Enable nginx site
  command: ln -s /etc/nginx/sites-available/roadtok8s-js /etc/nginx/sites-enabled
  args:
    creates: /etc/nginx/sites-enabled/roadtok8s-js
- name: Remove default nginx site
  file:
    path: "{{ item }}"
    state: absent
  notify: restart nginx
  with_items:
    - /etc/nginx/sites-enabled/default
    - /etc/nginx/sites-available/default

handlers:
- name: reload supervisor
  command: "{{ item }}"
  with_items:
    - supervisorctl reread
    - supervisorctl update
    - supervisorctl restart roadtok8s-js
  notify: restart nginx
- name: restart nginx
  systemd:
    name: nginx
    state: restarted

```

Now that we have our Ansible playbook complete, let's update our GitHub Actions workflow.

GITHUB ACTIONS WORKFLOW FOR NODE.JS AND ANSIBLE PLAYBOOK

Copy the entire GitHub Actions workflow file from the Python project at `~/dev/roadtok8s/py/.github/workflows/run-ansible.yaml` to the Node.js project at `~/dev/roadtok8s/js/.github/workflows/run-ansible.yaml`. After you do this, you'll need to change one line to make our new Ansible playbook work: `ansible-playbook deploy-python.yaml` to `ansible-playbook deploy-js.yaml`. Everything else should work exactly the same.

With this change, we can now commit our changes to Git and push them to GitHub. Before you can run the workflow, you need to verify the following GitHub Actions secrets:

- `SSH_PRIVATE_KEY`
- `AKAMAI_INSTANCE_IP_ADDRESS`

As a reminder, if you haven't done it already, you will need to create a new `SSH_PRIVATE_KEY` with `ssh-keygen`, install it on Akamai Linode, and then add the private key to GitHub Actions secrets. You will also need to create a new virtual machine on Akamai Linode with the new SSH key to obtain a new and valid IP address that you will add to GitHub Actions secrets as `AKAMAI_INSTANCE_IP_ADDRESS`.

Once you do that, you can run your GitHub Actions workflow and verify that your Node.js application is running by visiting the `AKAMAI_INSTANCE_IP_ADDRESS` after the workflow completes. I will leave it to you to go through this process and verify it works.

Summary

- Continuous integration is the process of constantly integrating code changes into a single source of truth, namely a Git repo.
- Continuous delivery is the process of constantly delivering code changes to a Git repo in preparation for deployment.
- Continuous deployment is the process of constantly deploying code changes to a production environment.
- Deploying consistent environments for your applications is essential to ensure your applications will work as intended.
- GitHub Actions workflows are designed to be easily understood and easy to use while still being incredibly flexible to the various tools we may need to use and automate.
- Ansible is a powerful tool that can be used to automate the installation and configuration of software on remote machines.
- NGINX is a powerful web server that sits in front of web applications to provide additional security and performance.
- Supervisor is a powerful process manager that can be used to ensure that our applications are always running.
- The YAML format is a staple of automation and deployment regardless of what programming language you write your applications in.
- Running applications on remote machines requires the systems to be continually updated and monitored to ensure the correct dependencies are installed and configured.
- Even with simple application design, the process to deploy applications can be complex and error prone.
- Node.js and Python runtimes can share a number of production dependencies, such as NGINX and Supervisor, but are different enough that they require specific configurations, causing more complexity and more potential for errors.

Containerizing applications



This chapter covers

- Installing and using basic Docker commands
- Running container images through the container registry
- Making applications portable and runtimes version-controlled
- Understanding hosting containers in registries
- Building, running, and pushing containers using a local machine

Back in the late 1900s, we had to insert disks and cartridges into our devices to run software or play games. Continuously updating these applications was mostly impossible, so developers worked tirelessly to ensure these applications were locked and as bug-free as possible before their release. As you might imagine, this was a painfully slow process, especially compared to today's standards.

These disks and cartridges could only be played or run on the devices they were designed for; Mario was played on Nintendo devices, and Windows computers ran Microsoft Word. In other words, the storage devices holding the software were tied directly to the operating system that could run the software. We still have the digital version of this today with app stores and digital downloads, with one major difference: the internet.

As you know, there are very few applications out today that do not use the internet in some capacity, regardless of the device they run on. This means that the applications we use every day are rarely tied to the operating system they run on. In other words, the hardware and OS matters less than it ever has before simply because applications use the internet and apps run on servers elsewhere. The first few chapters of this book highlighted this by the mere fact that I omitted rigorous system requirements for you to run Python, Node.js, Git, and so on. With late 1990s software, I would have had to give you a substantial list of hardware and system requirements to run the applications and maybe even write different book versions based on the OS you had. Now, I can just vaguely give you an OS, and there is a great chance the software will run on your machine.

That is, until it doesn't. Open source software tends to do very well with the initial installation process, but as soon as we start needing third-party dependencies, our applications and CI/CD pipelines can start to crumble, requiring a lot of attention to bring them back to a stable state. This is where containers come in.

Third-party packages are often essential to modern application development. For one, you might need to connect your app to a database of some kind. While there is likely a native Python or Node.js package you could use, those packages often assume system-level installations as well. You know what they say about assumptions, right?

Containers bundle your code and third-party system-level dependencies together so that you can have a portable application. This bundle is known as a *container image* and was pioneered and popularized by Docker. The magical thing about containers is that if a container runtime, also known as a Docker runtime, is installed on any given system, there's a great chance that your application will run without additional installations.

Containers still require a system-level dependency to run, a container runtime, but what's in the container could be Python, Node.js, Java, Ruby, and virtually any software that runs on Linux. Containers are the best modern solution for ensuring your app is portable and can run on any system with a container runtime installed. Containers have a number of limitations that we'll continue to explore for the remainder of the book. (Spoiler alert: One of these limitations is why Kubernetes exists.) Let's get started by installing Docker Desktop on your machine.

5.1 *Hello World with Docker*

For the purposes of this book, we will use Docker to build, run, and push our container images. There are a number of ways to do each of these tasks, but we'll stick to Docker to keep things simple.

Docker is most easily installed on Linux, macOS, and Windows Subsystem for Linux (WSL) with a tool called Docker Desktop. Windows Pro is required for Docker Desktop if you are not using WSL. To install Docker Desktop, follow these steps:

- 1 Visit the official Docker Desktop installation guide (<https://docs.docker.com/desktop/install/mac-install/>).

- 2 Follow the instructions for your operating system.
- 3 Open your command line and verify Docker is installed with `docker -- version`.

Once you have Docker installed, we'll see how simple it is to run third-party applications.

5.1.1 Running your first container

Docker has a lot of prebuilt container images we can run at any time. You can find a lot of open-source container images on Docker Hub (<https://hub.docker.com/>). Later in this chapter, we'll use Docker Hub to host our container image as well. Now let's run the official Docker-endorsed container image for NGINX (https://hub.docker.com/_/nginx) as seen in the following listing.

Listing 5.1 Hello World with Docker: Running the NGINX container image

```
docker run nginx
```

Figure 5.1 shows the output of running the NGINX container image.

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
3ae0c06b4d3a: Pull complete
efe5035ea617: Pull complete
a9b1bd25c37b: Pull complete
f853dda6947e: Pull complete
38f44e054f7b: Pull complete
ed88a19ddb46: Pull complete
495e6abbed48: Pull complete
Digest: sha256:08bc36ad52474e528cc1ea3426b5e3f4bad8a130318e3140d6cfe29c8892c7ef
Status: Downloaded newer image for nginx:latest
```

Figure 5.1 First container run output

Before we visit the running instance of this container image, we see the output that contains *Unable to find image . . .* and *Pulling from*, which signifies that Docker automatically attempts to download an image because it's missing from our local system. This download attempts to use Docker Hub, which is the official Docker container hosting, also known as a container registry. Docker will use this registry by default unless you have a good reason to change it. Docker registries are also open source, so you could, if you are super adventurous, host your own registry, but that's outside the context of this book.

For official Docker images, the command is `docker run <imageName>`, and third-party commands (including our own), the command is `docker run <userRegistryName>/<imageName>`, such as `docker run codingforentrepreneurs/try-iac`. We'll cover this in more detail later in this chapter.

So, where can we visit our NGINX instance? Well, we can't. Not yet, at least. We need to provide Docker with additional arguments, called flags, to be able to visit the application that the container is running.

5.1.2 Exposing ports

Running a container image with no arguments will run the software within the image, but it will *not* allow external traffic of any kind by default. To fix this, we need to expose the container images to a *port* on our local machine.

Ports are a way to allow traffic to enter and exit a system or, in our case, a running container image. We can pick and choose a port number for the system side, but we must respect the port number the application within the container is expecting. For NGINX, the default configured port is 80. For our local system, we'll use the port of 8080 to help distinguish the command. The syntax for this argument, also known as a flag, is `-p <localPort>:<containerPort>`, which translates to `-p 8080:80` for this example. Here's a few examples changing the port numbers:

- `docker run -p 8080:80 nginx`—Our localhost:8080 maps to the NGINX container port 80.
- `docker run -p 6543:5432 postgres`—Our localhost:6543 maps to the Postgres container port 5432 (https://hub.docker.com/_/postgres).
- `docker run -p 3001:8080 codingforentrepreneurs/try-iac`—Our localhost:3001 maps to the try-iac container port 8080.

Assuming you have `docker run nginx` still running in your terminal, press Control + C to stop that running container and start a new one at `docker run -p 8080:80 nginx`. Once you do that, visit `http://localhost:8080` on your local machine, and you should see the standard NGINX Hello World, just like in figure 5.2.



Figure 5.2 Visiting port 80 in the NGINX container image on localhost:8080

Exposing ports as one of our first steps helps highlight that containers are mostly isolated from the surrounding operating system environment until we add additional parameters to change how that isolation works. In other words, containers are designed to be isolated from the host computer by default. This is a good thing because it means we can run virtually any software in a container without touching our system's

configuration other than to run Docker itself. The container has its own world within it as well. Let's see what I mean by entering the container's bash shell.

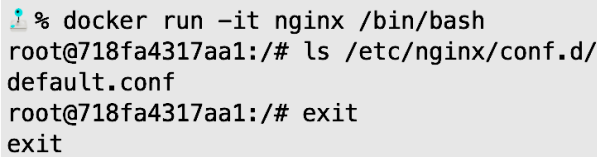
5.1.3 Entering a container

Containers have their very own shell within the container, along with a file system and almost everything a normal operating system has. This is why containers are often referred to as *lightweight virtual machines* because they are isolated from the host operating system but still have a lot of the same functionality.

Let's look at what's in the NGINX container by using another flag that allows us to use the interactive terminal `-it`. This interactive terminal is much like a Secure Shell (SSH) connection that we have used in previous chapters. To use the container's bash shell, we will use the interactive terminal. If we need to run other commands (e.g., user input commands from an application), we could also use the interactive terminal (e.g., the container-based Python or Node.js shell). To enter our NGINX container, run the following commands:

- `docker run -it nginx /bin/bash`—Enters the Bash shell
- `ls /etc/nginx/conf.d/`—Lists the files in the NGINX configuration directory
- `exit`—Exits the container

You should see something that resembles figure 5.3.



```
% docker run -it nginx /bin/bash
root@718fa4317aa1:/# ls /etc/nginx/conf.d/
default.conf
root@718fa4317aa1:/# exit
exit_
```

Figure 5.3 Entering the NGINX container image's bash shell

As we see from this simple example, the container runtime has a world of its own that may or may not exist on our localhost. This is why containers are so powerful. We can run virtually any software in a container, and it will run the same way on any system with a container runtime installed. You might be tempted to treat a container like a virtual machine, and you can, but I'll offer a simple word of warning: *containers are designed to be destroyed*.

5.1.4 Stateless containers

Database applications are great examples of software that has a state. If you destroy the machine running the database, there's a good chance that you'll lose all of your data. The *state* of the data needs to persist somewhere, so having backup data is essential for stateful applications. Virtual machines, just like your local machine, are stateful by default, but if the machine is destroyed, the state, or data, is gone forever.

Our Python and Node.js applications are currently examples of *stateless* applications because we can always recreate the same environment regardless of whether the running machine is destroyed or not. As you are hopefully aware, recreating the environment is possible and mostly easy thanks to the `requirements.txt` and `package.json` files.

Before we dive into the stateless nature of containers, let's work through a practical example by running a Linux-based container image and installing NGINX. The purpose of this example is to help illustrate what we *should not do* with containers but *should do* with virtual machines.

For this example, we'll add the flag `--rm` to our `docker run` command to immediately remove the container after it's stopped. Here's what we'll do:

- 1 `docker run --rm -it ubuntu /bin/bash`—Downloads ubuntu and enters the bash shell environment
- 2 `apt-get update`—Updates the package manager within the container
- 3 `apt-get install nginx`—Installs NGINX
- 4 `cat /etc/nginx/sites-available/default`—Shows the NGINX configuration file
- 5 `nginx -v`—Shows the NGINX version
- 6 `exit`—Exits the container

This installation process feels so similar to what you might expect within a virtual machine environment because this container is still Ubuntu! So, at this point, we should be good to go with NGINX, right? Well, let's just use *bash shell* on this container again and see what happens:

```
docker run --rm -it ubuntu /bin/bash
cat /etc/nginx/sites-available/default
nginx -v
exit
```

Wait what? Errors! Why?! Doing this process again without installing NGINX gives us the errors `No such file or directory` and `command not found`. These errors tangibly show us that we cannot long-term modify an already existing container image from *within* the container environment. This is the stateless nature of containers. So, how do we modify containers? This is the purpose of a Dockerfile and what we'll do in the next section.

5.2 *Designing new container images*

Creating a new container image requires at least two things: a container builder and a Dockerfile. The *container builder* is just another command that is available to us by using Docker Desktop. The Dockerfile is a special document that allows us to write instructions for how to build our container image.

Thinking back to the last section, we'll create our first container image by installing NGINX directly to the pre-existing Ubuntu container image. This will allow us to modify the container image and save it as a new image.

5.2.1 Creating a Dockerfile

The syntax for a Dockerfile is unique to Docker, but it usually boils down to using the following commands:

- `FROM <image>`—The prebuilt container image we can inherit from.
- `RUN <your command>`—An instruction to install other software or run a command. I tend to think of this as a bash shell command that is cached during builds (because that's what it does). You'll use this one a lot.
- `CMD <your command>`—An instruction to run a command when the container is started. For NGINX, this command will be `nginx -g 'daemon off;'`. The instructions are broken up into an array of strings, so it results in `CMD ["run", "this", "--app"]` in place of just `run this --app`.

Let's create our first Dockerfile on our machine at the location `~/Dev/roadtok8s-hello-docker/Dockerfile` with the contents from the following listing.

Listing 5.2 Your first Dockerfile

```
FROM ubuntu
RUN apt-get update && apt-get install -y nginx
CMD ["nginx", "-g", "daemon off;"]
```

Before we use this Docker, let's name this container simply `rk8s-nginx`. We'll use this name when we build the container through the tag flag: `-t <your-tag name>`. Now we need to navigate to the directory of your Dockerfile and build and tag our container image as seen in listing 5.3.

Listing 5.3 Building your first container Image from a Dockerfile

```
cd ~/Dev/roadtok8s-hello-docker
docker build -t rk8s-nginx -f Dockerfile .
```

← The directory where your Dockerfile should be located

Figure 5.4 shows the output of running the build command. The time to build this container image will vary from machine to machine, but it should be relatively quick because our Dockerfile makes very few changes to the prebuilt Ubuntu container image we're inheriting from.

```
% docker build -t rk8s-nginx -f Dockerfile .
[+] Building 0.0s (6/6) FINISHED                                docker:desktop-linux
=> [internal] load .dockerignore                                0.0s
=> => transferring context: 2B                                    0.0s
=> [internal] load build definition from Dockerfile              0.0s
=> => transferring dockerfile: 131B                               0.0s
=> [internal] load metadata for docker.io/library/ubuntu:latest 0.0s
=> [1/2] FROM docker.io/library/ubuntu                          0.0s
=> CACHED [2/2] RUN apt-get update && apt-get install -y nginx  0.0s
=> exporting to image                                           0.0s
=> => exporting layers                                           0.0s
=> => writing image sha256:f77b47102f215d4e27878d1d5c232c000f82915fc0f33c0779b 0.0s
=> => naming to docker.io/library/rk8s-nginx                    0.0s
```

Figure 5.4 Building your first container image from a Dockerfile

If this is your first time building a container image, congratulations! Now let's run it and see what happens.

5.2.2 *Running your first container image*

Now that we built our first container image from a custom Dockerfile, let's run it. The important piece to remember is that we tagged it `rk8s-nginx` on our local machine, so we'll use that name to run it. Listing 5.4 shows the command to run our container image.

Listing 5.4 Running your first container image from a Dockerfile

```
docker run -p 8080:80 rk8s-nginx
```

Once you run this, you should be able open <http://localhost:8080> to view the standard NGINX Hello World page once again. If the port is in use, you can change the port number to anything you'd like or stop the application/container running on that port.

Once again, we'll verify that NGINX was installed on our container through the bash shell. Run the following commands:

```
1 docker run --rm -it rk8s-nginx /bin/bash
2 cat /etc/nginx/sites-available/default
3 nginx -v
4 exit
```

All of these commands (we swapped `ubuntu` for `rk8s-nginx`) should now be available in our modified version of Ubuntu thanks to our Dockerfile and built container! Now that we have the basics down, let's create our Python and Node.js container images.

5.3 *Containerizing Python applications*

Docker Hub provides a large number of prebuilt and official container images, like the [https://hub.docker.com/_/python\[official Python\]](https://hub.docker.com/_/python[official Python]) container image. Just like with using the https://hub.docker.com/_/ubuntu container image, prebuilt images make our lives much simpler.

Containerizing Python applications is very approachable because the Python Package Index (PyPI; <https://pypi.org/>) allows us to use *pip* to install tools like FastAPI, Django, requests, and gunicorn. Without PyPI, we could still install tools, but it would be a much more involved process. With this in mind, let's create our Python Dockerfile.

5.3.1 *Version tags and a Python Dockerfile*

When it comes to Python projects, we typically start with a specific version in mind. Previously, we mentioned we must use Python 3.8 or higher. In this section, we'll use Python 3.11 as our version of choice, but how?

On the one hand, we could use the `RUN <command>` block in our Dockerfile to install a specific version. On the other hand, the Docker Hub Python container image has a large number of tags (https://hub.docker.com/_/python/tags) that include many versions of Python.

To find the version we need, we'll need to do the following steps:

- 1 Visit the official Python container repo at https://hub.docker.com/_/python.
- 2 Click the Tags tab (or navigate directly to https://hub.docker.com/_/python/tags).
- 3 Filter the results by 3.11.4 (or whatever version you want).

Figure 5.5 shows the results of the previous steps with a few versions cropped out.

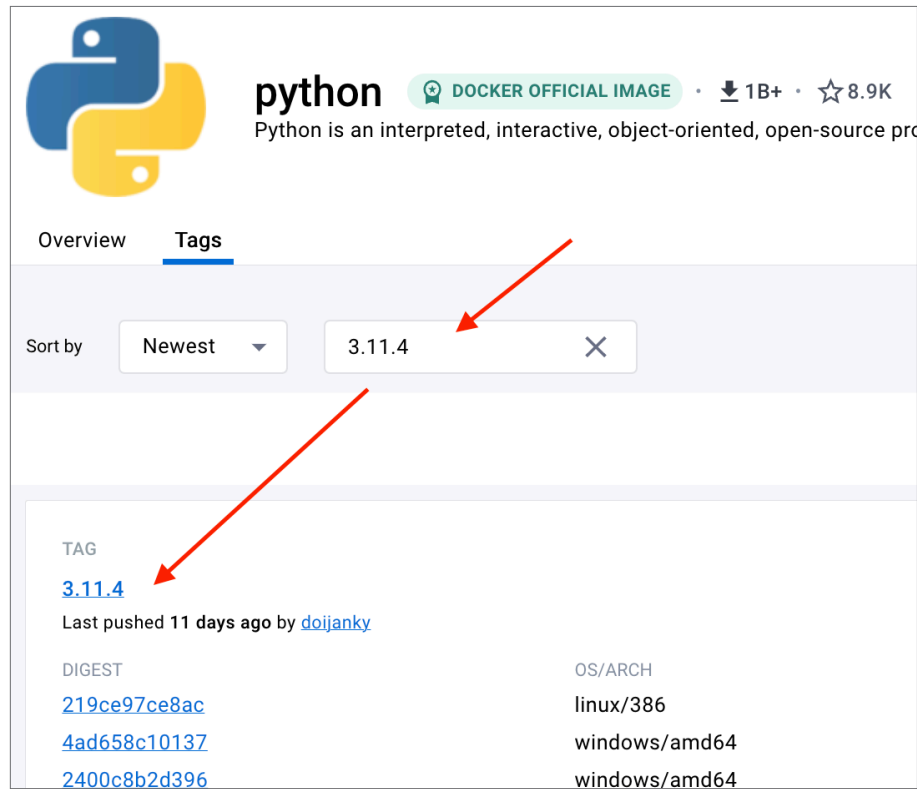


Figure 5.5 Docker Hub Python tags

Figure 5.6 shows the exact command to *download* the image to your local machine. This process, known as pulling an image, is done automatically when we run or build an image.

Docker Hub maintains nearly *all versions* of stable code that has been released for major open-source projects. A huge benefit to this is that, in 2035 and beyond, version 3.11.4 will probably still be available exactly as it is here. If it's not, that is likely due to some major security risk or a massive change to Docker Hub.

```
docker pull python:3.11.4
```



Figure 5.6 Docker Hub screenshot of the pull command

In this case, the `docker pull python:3.11.4` command shows us the exact container tag we need to use this image. The format is `docker pull <containerName>:<tag>`. We can use this format in our Dockerfile in the `FROM <image>` line, as we'll see in the next section.

Before we carry on, let's run this container image locally to see what happens. The following listing shows the command to run this container image.

Listing 5.5 Running the Python container image

```
docker run -it python:3.11.4
```

When you run this, there's a good chance the container will need to download (in my case, I saw `Unable to find image 'python:3.11.4' locally`). Once it finishes downloading, I have the Python shell at my fingertips running through Docker! Figure 5.7 shows the output of running this command.

```
% docker run -it python:3.11.4
Unable to find image 'python:3.11.4' locally
3.11.4: Pulling from library/python
42cbebf8bc11: Pull complete
9a0518ec5756: Pull complete
356172c718ac: Pull complete
dddc3ceb998: Pull complete
8e07adfdac1d: Pull complete
dc87587bf469: Pull complete
b7ee9aadd410: Pull complete
47d5195c2be5: Pull complete
Digest: sha256:d73088ce13d5a1eec1dd05b47736041ae6921d08d2f240035d99642db98bc8d4
Status: Downloaded newer image for python:3.11.4
Python 3.11.4 (main, Jul 4 2023, 21:57:59) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Figure 5.7 Running the Python container image

Take a moment to appreciate how fast this works *and* how easy it is to use nearly *any* version of Python within a few minutes (if not seconds). This is the power of containers. Using nearly any version of nearly any open-source programming language from the past decade can run within minutes of you choosing to do so. Do you remember how long it took to install just Python? Then Node.js? What if you need to use Ruby? Or Java? Or Go? Or Rust? Or C++? Or C? Or PHP? Or Perl? Or . . . you get the point. As long as Docker is installed, nearly any version of nearly any open-source programming language can be installed within minutes.

With this power, we can also run nearly any version of any iteration of our own applications within minutes (if not seconds). This paradigm is what excites me about containers. Let's see it in action by containerizing our Python application.

5.3.2 Creating a Python Dockerfile

Now that we can run a specific version of Python with Docker, we will start the process of designing our application for our specific Dockerfile.

The final steps we'll take are nearly identical to what we did with Ansible in chapter 4 but modified for the Dockerfile syntax. Here's a recap of what our Dockerfile needs to do:

- Copy our project code.
- Install OS-level dependencies such as `python3-venv`, `python3-dev`, and `python3-pip`.
- Create a virtual environment.
- Update the virtual environment's default pip installation.
- Install our project dependencies into the virtual environment.
- Run our app.

Two major packages are purposefully absent: NGINX and Supervisor. We'll see why these packages are no longer needed very soon.

Before we run our code, we're going to make a Dockerfile that is *almost* complete so we can see what is available to us by default. This means we will use the following commands:

- Change our working directory to `/app` (this will become clear within the Dockerfile).
- Copy our `src/` folder to the `/app` directory.
- Run `apt-get update && apt-get install -y python3-venv python3-dev python3-pip`. This will ensure the OS-level dependencies are installed within our container.
- Add `python3 -m venv /opt/venv`. We will store our virtual environment in the `/opt/` location, which is a recommended place to store add-on software like virtual environments.
- Add `/opt/venv/python -m pip install --upgrade pip`. It's always a good idea to update our virtual environment pip installation *before* we install our project requirements.
- Run `/opt/venv/python -m pip install -r /app/src/requirements.txt` to install our project requirements.
- Run `/opt/venv/python -m http.server 8080`. This will run a built-in Python web server that will either render an `index.html` file, if one exists, or display the contents of any given folder; we'll have the latter.

We have used each of these commands a few times throughout this book, with the exception of `python -m http.server`. The goal now is not to introduce too many new commands but just reframe how we use commands we are starting to become familiar with.

With this in mind, navigate to the root of your Road to Kubernetes Python project. If you have been following along, the project will be located at `~/dev/roadtok8s/py`. In this project we'll create a `Dockerfile` with the contents in the following listing.

Listing 5.6 Python Dockerfile for browsing

```
# get our Python image
FROM python:3.11.4

# create and move to the /app directory
# all commands after will execute in this directory
WORKDIR /app

# copy our local src folder to /app in the container
COPY ./src/ /app

# Run os-level updates
RUN apt-get update && \
    apt-get install -y python3-venv python3-dev python3-pip

# Create our Python virtual environment
RUN python3 -m venv /opt/venv

# Update the virtual environment pip
RUN /opt/venv/bin/python -m pip install --upgrade pip

# Install our project dependencies
# src/requirements.txt is our local version
RUN /opt/venv/bin/python -m pip install -r requirements.txt

# Run a server to expose copied files
CMD ["/opt/venv/bin/python", "-m", "http.server", "8080"]
```

Now let's build our container with the commands in the following listing.

Listing 5.7 Building and running the Python Dockerfile

```
cd ~/dev/roadtok8s/py
cat Dockerfile # should yield correct values

# now build the container
docker build -t rk8s-py -f Dockerfile
docker run -p 8080:8080 rk8s-py
```

Use any tag you see fit.

Port 8080 is exposed in the Dockerfile.

With this built and running, let's open our browser to `http://localhost:8080`, and we should see the same results as figure 5.8.

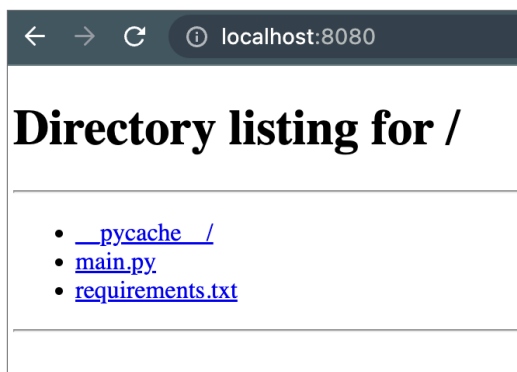


Figure 5.8. Browsing the Python Dockerfile output

What we see here are a few important pieces of information:

- `__pycache__/` was copied.
- Our working directory contains exactly what is in our local `src/` folder.
- Python's built-in `http.server` can be used to serve static websites if we need to by simply adding an `index.html` file to our `src/` folder.
- Since we have the `http.server` running, as many Python developers already know, we can use `gunicorn` in its place. We can, but we'll address that soon.

Before we complete this Dockerfile, we must address the problem of extra files we do not want to include. This is where a `.dockerignore` file comes in handy.

5.3.3 Ignoring unwanted files in Docker builds

Docker has two commands for adding files or folders to a container image during the build phase: `COPY` and `ADD`. `COPY` is the same as `ADD`, except `ADD` can handle remote URLs to download files from the internet.

These commands will copy everything into your Docker image, which is rarely what we want. Instead, we can use a special format nearly identical to *Git's* `.gitignore` file called `.dockerignore`. This file will allow us to ignore files and folders from being copied into our container image.

At this point, let's make a basic `.dockerignore` next to our Dockerfile that contains the single line: `**/__pycache__/` and see if that folder is still included in our builds.

After you build this file, you should see that the `__pycache__/` folder is no longer present. So, to ignore files, you can use a `.dockerignore` with the following rules in mind:

- Use absolute paths for individual files.
- Use relative paths for folders prepended with `**/` to ignore all files within that folder.
- Ensure folders end with `/` to ignore all files in the stated folder.

Once our build phase is moved to GitHub Actions, we will likely have less of a need for a `.dockerignore` simply because then we'll be using both Git and a `.gitignore` file to push the code. In other words, the files and folders you would typically want Docker to ignore should already be absent. That said, I always recommend using a `.dockerignore` to make your final built container image lighter (e.g., fewer files and folders) to decrease the build time, decrease the deploy time, and decrease the likelihood of exposing a file you should *not* expose (e.g., any `.env` files or `.secret` files). The following listing shows the output of our container image after adding a `.dockerignore` file.

Listing 5.8 Python `.dockerignore` sample

```
# ignore Python cached files
**/__pycache__/*
*.py[cod]

# ignore env files
*.env
```

Now that we have a `.dockerignore` file, let's move to the final step of our Dockerfile: creating a script to boot our application.

5.3.4 *Creating and running an entrypoint script*

Now we need to actually run our application within the Docker container. You may recall we previously used Supervisor to run the application for us. We're going to be using essentially the same command, just stored in a different location.

Before we do, let's talk about the final line in a Dockerfile: `CMD <your command>`. We have two approaches we can take here: directly running our application or running a script that runs our application. For Python applications, I prefer to use a script to run our application. This script is called an *entrypoint* script. Either way, the `CMD` directive will be executed at runtime and not during the container build phase; more on this soon.

I prefer running an entrypoint script because it allows us to add additional logic to our application before it runs. For example, we could add a step to check if a database is available before running our application.

An entrypoint script should not be confused with the `ENTRYPOINT` directive in a Dockerfile. `CMD` can be swapped for `ENTRYPOINT`, but we'll stick to `CMD` for now for simplicity's sake; `ENTRYPOINT` is better suited for non-web-app applications.

Our entrypoint script is going to be based on the following conditions:

- `export RUNTIME_PORT=${PORT:-8080}`—This will set the `RUNTIME_PORT` environment variable to the value of the environment variable `PORT` if it exists or `8080` as a fallback.
- `/opt/venv/bin`—The location of our various virtual environment commands.
- `/opt/venv/bin/gunicorn`—The location of our gunicorn executable based on the virtual environment and the `RUN ... pip install` step of our Dockerfile.

- `--worker-class uvicorn.workers.UvicornWorker main:app --bind "0.0.0.0:$RUNTIME_PORT`—The arguments to gunicorn to run our application. This is the same command we used in chapter 4, but with a few changes:
 - We no longer need a process ID; the running Docker image will be our process.
 - The gunicorn port is now bound to an environment variable, offering flexibility outside the container itself.

As a reminder, environment variables are where you want to store sensitive runtime information, such as passwords and API keys. Environment variables can also be used to modify the configuration of the application runtime (e.g., `PORT`), which is why our entrypoint has a fallback to a `PORT` environment variable. Sensitive environment variables should *never* be stored in the codebase, added to a `Dockerfile`, or included in a container build (e.g., using `.dockerignore` to ignore common environment variable files like `.env` or `.secrets`). Most container runtimes, like Docker or Kubernetes, make it easy to inject environment variables when we run the container. With this in mind, let's create a new file in our `src/` folder called `conf/entrypoint.sh` with the contents in the following listing.

Listing 5.9 Python entrypoint script

```
#!/bin/bash
export RUNTIME_PORT=${PORT:-8080}

/opt/venv/bin/gunicorn \
  --worker-class uvicorn.workers.UvicornWorker \
  main:app \
  --bind "0.0.0.0:$RUNTIME_PORT"
```

A key trait of the *location* of this entrypoint is that it is not stored in `src/`, but rather in `conf/`, right next to `nginx.conf` and `supervisor.conf`. I prefer to keep all of my configuration files in a single location; this is no different, but it also gives us the opportunity to once again use the `COPY` command for a specific file.

This *entrypoint* script will be the final command we run, which means the script itself needs to have the correct permissions to be executed (e.g., `chmod +x`); otherwise, we'll get a permission error after we build this container when we attempt to run it. The following listing shows the updated lines for our `Dockerfile`.

Listing 5.10 Final Python Dockerfile

```
...
COPY ./src/ /app

# copy our local conf/entrypoint.sh to /app in the container
COPY ./conf/entrypoint.sh /app/entrypoint.sh
...
# Make our entrypoint script executable
RUN chmod +x /app/entrypoint.sh
...
# Execute our entrypoint script
CMD ["/app/entrypoint.sh"]
```

Let's go ahead and test our new Dockerfile by building and running it:

- *Build it*—`docker build -t rk8s-py -f Dockerfile`.
- *Run it*—`docker run -p 8080:8080 rk8s-py`.
- *Run it with a new PORT environment variable*—`docker run -p 8080:8081 -e PORT=8081 rk8s-py`. Notice that the *internal container port* (`-p <local-port>:<container-port>`) matches the environment variable port value declared with `-e PORT 80801`.
- *Open another terminal and run it again with another new PORT value*—`docker run -p 8081:8082 -e PORT=8082 rk8s-py`.

Depending on the production hosting choice for our container images, the `PORT` variable is often set for us, so we need to ensure our application is flexible enough to handle this. Now that we have our Python container image built, let's move to our Node.js container image.

5.4 Containerizing Node.js applications

It might come as little surprise that the process for containerizing a Node.js application is nearly identical to a Python application—by design. In our case, we're going to containerize our Express.js app, which has a standard Node.js runtime. It's important to note that not all Node.js web applications *use* the Node.js runtime as their web server. For example, server-side rendered (SSR) applications do not always use the Node.js runtime as their web server (e.g., React.js can be an SSR app). The process goes like this:

- 1 Find the Node.js version tag we need in the Node.js image repo on Docker Hub (http://hub.docker.com/_/node).
- 2 Use the `FROM` directive to use this image and version tag.
- 3 `COPY` our code into our `WORKDIR`.
- 4 `RUN` the `npm install` command to update `npm` and install our `package.json` project dependencies. (As a reminder, there are other package managers for Node.js, but we'll stick to `npm` for this book.)
- 5 Write an entrypoint script to manage our application's runtime and default `PORT`.
- 6 Create the `CMD` directive to run our entrypoint script.

To find the version we need, we'll need to do the following steps:

- 1 Visit the official Node container repo at https://hub.docker.com/_/node.
- 2 Click the Tags tab (or navigate directly to https://hub.docker.com/_/node/tags).
- 3 Filter results by `18.17.0` (or whatever version you want).
- 4 Sort results by `A-Z`.

Figure 5.9 shows the results of these steps.

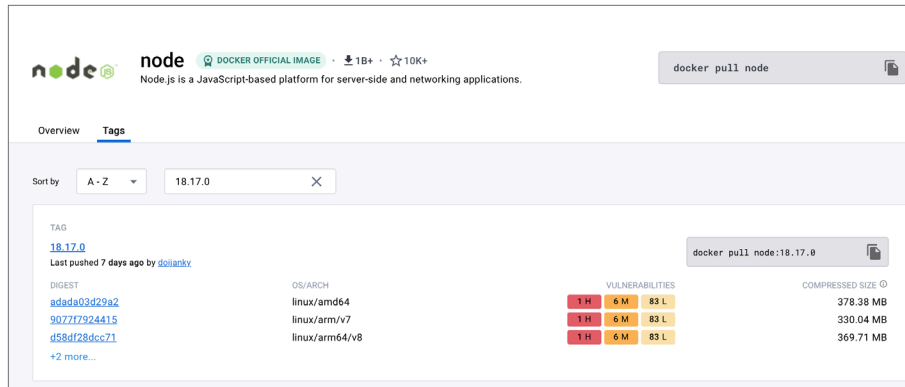


Figure 5.9 Docker Hub Node.js tags

Figure 5.10 is a screenshot of the command direction from the Node.js repo to pull the image and specific tag from Docker Hub. It is important to recognize what to look for to ensure you are using the correct tag for any given container image.

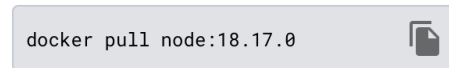


Figure 5.10 Docker Hub node pull command

Once again, `docker pull node:18.17.0` shows us the exact container tag we need to use for this image.

Now that we have the container image and image tag we need, let's create the entrypoint script we need prior to creating our Dockerfile.

5.4.1 The Node.js entrypoint script and .dockerignore file

As you may recall from previous chapters, we run our Express.js application directly with Node.js with the command `node main.js`. Within our `main.js` file, we also have the code for an environment-variable-based `PORT` value (e.g., `const port = process.env.PORT || 3000;`). This means that our entrypoint script is going to be pretty simple:

- Determine the location of `main.js`.
- Run `node path/to/main.js`. Like we did with the Python app, we will use the `/` app folder. Unlike the Python process, we will also use the `/src` folder to maintain relative paths.

The following listing shows the contents of our entrypoint script located at `conf/entrypoint.sh` in our Node.js application (located at `~/Dev/roadtok8s/js` if you have been following along).

Listing 5.11 Node.js entrypoint script

```
#!/bin/bash

node /app/src/main.js
```

As we can see, the Node.js entrypoint script is incredibly simple but also very consistent with how we containerized our Python application.

Our final pre-Dockerfile step is to create a `.dockerignore` file that includes all the files and folders we want to ignore. The following listing shows the contents of our `.dockerignore` file located at `~/Dev/roadtok8s/js/.dockerignore`.

Listing 5.12 Node.js Dockerignore

```
# ignore node modules
**/node_modules/

# ignore env files
*.env
```

Now that we have all the code we need, let's create our Dockerfile.

5.4.2 The Node.js Dockerfile

As a first step to creating another Dockerfile, let's re-examine the various actions we will need to meet to build our container:

- Create a working directory (`WORKDIR`) for our project; we'll use the commonly used `/app`.
- Copy all code from `src/` to `app/src/`. With our Node.js app, we want to maintain paths relative to the root of our project due to the fact that the `node_modules/` will be installed in the root of our project. Moving `node_modules` is not as easy as moving a Python virtual environment and thus is not recommended.
- Copy both `package.json` and `package-lock.json` files. `package-lock.json` will help ensure consistent builds from local to container.
- Copy `conf/entrypoint.sh` to `app/entrypoint.sh`.
- Run `npm install` to install our project dependencies.
- Run `chmod +x entrypoint.sh` to ensure our entrypoint script can be executed.
- Write the `CMD` directive to run our entrypoint script.

All of these steps should feel repetitive for a good reason: we are repeating almost the same process. Honing in on a consistent environment often means repeating ourselves consistently. The following listing shows the final Dockerfile for our Node.js application.

Listing 5.13 Node.js Dockerfile

```
# Using the official Node.js image for the LTS version
FROM node:18.17.0

# Set our app's working directory
WORKDIR /app

# copy the primary source code
COPY ./src /app/src/

# copy the package.json and package-lock.json to
# ensure we use the same dependencies
COPY package.json /app/
COPY package-lock.json /app/

# copy the entrypoint.sh for running the app
COPY conf/entrypoint.sh /app/

# install dependencies
RUN npm install

# make the entrypoint.sh executable
RUN chmod +x entrypoint.sh

# run our app
CMD ["/entrypoint.sh"]
```

Now that we have this Dockerfile, let's run and build it with the following steps:

- 1 Build the image with `docker build -t rk8s-js -f Dockerfile`.
- 2 Run the image with `docker run -p 3000:3000 rk8s-js`. If you recall, we set the default PORT within our Express.js app to 3000.
- 3 Try a new port: `docker run -p 3001:3000 -e PORT=3001 rk8s-js`.
- 4 Add a new environment variable: `docker run -p 3002:3000 -e MESSAGE="Hello World" rk8s-js`.

The output of these commands will be nearly identical to the Python containerization process, so I'll leave it to you to verify. Now that we know how to pull, build, and run container images, let's learn how to upload them to Docker Hub using the built-in push command.

5.5 Pushing container images to Docker Hub

Before we can start distributing our built container images, we need to host them on a Docker registry. For the remainder of this book, we will use Docker Hub as our registry of choice to keep things as simple as possible.

Docker registries store and host container images, much like Git repositories store and host code. Docker registries are open source, so you could create your own registry to host your containers, but doing so is a *lot* more complex than what we cover in this

book. Docker Hub is the official Docker registry and is what the `docker` command line tool uses by default. Here are the steps for using Docker Hub:

- 1 Create an account on <http://hub.docker.com> and get a username; mine is *codingforentrepreneurs*.
- 2 Log in to Docker Hub from the command line with `docker login`.
- 3 The steps to build and push, which we'll cover this more in a moment:
 - `Build docker build -t <your-username>/<your-container-name>:<your-tag> -f <your-dockerfile>.`
 - `Push docker push <your-username>/<your-container-name>:<your-tag>.`
 - If you push to your Docker Hub account, a new public container registry will be created by default. You can also manually create one to pick a status for the container, public or private.

With this in mind, let's proceed to a build-to-push process for our Python application.

5.5.1 *Your first push to Docker Hub*

Pushing to Docker Hub is nearly as simple as building on a local machine. The following listing offers a complete look at how to navigate, build, and push our container to Docker Hub.

Listing 5.14 Pushing our Python container image to Docker Hub

```
cd ~/dev/roadtok8s/py

# if you haven't already
# login to Docker Hub
docker login

# build and tag twice
docker build -f Dockerfile \
  -t codingforentrepreneurs/rk8s-py:0.0.1 \
  -t codingforentrepreneurs/rk8s-py:latest \
  .

# push a specific tag
docker push codingforentrepreneurs/rk8s-py:0.0.1

# push all tagged versions
docker push codingforentrepreneurs/rk8s-py --all-tags
```

Before we review the output, I want to point out that I built the image once but tagged it two times using the flag `-t`. I did this because of a standard Docker feature. If you recall when we ran `docker run nginx`, we were actually running `docker run nginx:latest`; the `latest` tag is used by default if no tag is specified. Tagging my container build with `latest` and a specific version gives me the flexibility to use either tag while I am also storing iterations of any given version. This scenario is meant to highlight the

version-control nature of tagging, building, and pushing images to Docker Hub. Figure 5.11 shows the output of our push command.

```
% docker push codingforentrepreneurs/rk8s-py --all-tags
The push refers to repository [docker.io/codingforentrepreneurs/rk8s-py]
7193915edb5c: Pushed
2987fbbcb4df: Pushing [=====>] 19.47MB
0aef47a0d5e4: Pushing [=====>] 17.27MB
8e76126f26f7: Pushing [=====>] 28.74MB
3516bc901dcf: Pushing [=====>] 34.72MB/79.98MB
d3af68212aa8: Pushing [=====>] 2.56kB
1e70be2a822c: Waiting
73a330a800aa: Waiting
```

Figure 5.11. Pushing our Python container image to Docker Hub

This output shows all the layers that make up the built container being pushed to Docker Hub. These same layers should resemble the `docker run` or `docker pull` commands when we are using an image that isn't on our local machine.

The layers we see in figure 5.11 correspond roughly to each instruction (e.g., `FROM`, `COPY`, `RUN`, etc.) from our Dockerfile. Each instruction is cached and reused when we build our container image. This caching is also why our first build may take a while, but subsequent builds are sped up. This is also why Docker registries can host massive container images without taking up a lot of space or taking a long time to upload or download. With the build complete and pushed to Docker Hub, let's take a look at our repository on Docker Hub. Figure 5.12 shows the repository page for our Python container image.

The screenshot shows the Docker Hub interface for the repository `codingforentrepreneurs/rk8s-py`. The page has tabs for General, Tags, Builds, Collaborators, Webhooks, and Settings. The General tab is active, showing a description field, a 'Last pushed' timestamp of 6 minutes ago, and a table of tags. The tags table lists two tags: `latest` and `0.0.1`, both pushed 6 minutes ago. There is also a 'Recent builds' section and a 'Docker commands' section with a 'Public View' button.

Tag	OS	Type	Pulled	Pushed
latest	linux	Image	---	6 minutes ago
0.0.1	linux	Image	---	6 minutes ago

Figure 5.12 Docker Hub rk8s Python container image

This page shows us a few important pieces of information:

- Our container image is public by default (this is easy to change). If we wanted it to be private before our first push, we would create the repository first.
- We can see the two tags we pushed to Docker Hub (e.g., *latest* and *0.0.1*).
- If we go to the Tags tab, we can see the container size (not pictured, but it's about 400 mb).

If you're working on Apple Silicon or an ARM-based Raspberry Pi, you're going to need to add an additional flag to account for the platform. This process isn't a problem until it is, so let's make sure it isn't. If you are not on these platforms, feel free to skip the next section.

The Docker build platform

If you're on Apple Silicon (e.g., M1 Chip, M2 Chip) or Raspberry Pi, that means your system architecture is ARM, while most cloud-based Linux servers have an x86 architecture.

When you run `docker build`, it will natively build for the system architecture you are using, which can cause problems when you try to run this container on other platforms (e.g., a Linux virtual machine).

The solution is simple; you just need to include `--platform=linux/amd64` in your build command, such as using `docker build -f Dockerfile --platform=linux/amd64 <your-username>/<your-repo>:<your-tag>`.

There is another, newer Docker tool called *buildx*, which enables multi-platform, multi-architecture builds; this is outside the scope of this book because it's a bit more complex than what we need.

Instead of using multi-platform builds, we will build our container images on GitHub Actions. This will allow us to build our container images on a Linux-based virtual machine and push them to Docker Hub. This will also allow us to build our container images on every commit to our main branch, which will ensure our container images are always up-to-date.

At this point, we have done everything we need to do to build and run container images on our local machine. It's a good idea to continue to experiment with these local containers before we start automating the process of building and pushing them to Docker Hub. I want to emphasize that there is a *lot* more to Docker than we covered here, but understanding this chapter will be crucial for deploying a containerized application to production.

Summary

- Containers are mini-Linux machines, and we have a lot of flexibility in how we can use them.
- Containers unlock portability for nearly any kind of application, especially open-source ones.
- The pattern of instructions for containers uses a logical method of copying and installing the software any given application might need.
- Containers are stored in a container registry like Docker Hub so they can then be distributed to production systems.
- Building containers is essentially identical regardless of the application, except for what goes into the Dockerfile.
- The container is responsible for running the containerized application or applications through what's declared in the Dockerfile (or as an additional argument).
- Containers force consistency in how we package our final application.
- A containerized application can run differently based on the environment variables passed at runtime.

Containers in action

This chapter covers

- Using GitHub Actions to build and push containers
- Attaching storage volumes and maintaining state
- Using file-based environment variables
- Running and managing multiple containers with Docker Compose
- Learning container-to-container networking and communication basics

Imagine sitting in an audience where CodeOprah comes out and says, “You get a container and you get a container and you get a container—everyone gets a container!” The crowd goes wild. But you don’t—not yet, anyway. You’re sitting there asking yourself a few fundamental questions. How do I run a database within a container? How do my two containers talk to each other? How do I manage multiple containers at once? Before we start passing out containers like they’re free cars, we need to configure our end-to-end CI/CD pipeline to allow GitHub Actions to be responsible for building and pushing our containers to Docker Hub.

Once our build process is automated, we need to understand how to inject state into containers by using volumes. Volumes are a way to persist data directly in a container regardless of the current status or state of the container. They can exist even if the container is not running or completely destroyed. Using volumes is how we get stateful, or the opposite of stateless, containers.

Volumes are not the only way to enable state in containers. Environment variables can have API keys or secret credentials to external database services, thus enabling a form of application state.

Attaching volumes and using environment variables with the `docker` command can be simple to use but difficult to remember and manage. Let's look at a couple of examples:

```
■ docker run -v path/to/local/volume:/path/to/container/volume -e
  ➔ENV_VAR=VALUE --env-file ./path/to/env/.env -p 8000:8000 -d
  ➔my-container-image-a

■ docker run -v path/to/local/vol-b:/path/to/container/vol-b -e ENV_
  ➔VAR=VALUE -e PORT=9090 -e DEBUG=1 --env-file ./path/to/env/.dev-file
  ➔-p 8000:8000 -d my-container-image-b
```

Imagine trying to run this for more than one project? It's going to be a nightmare to remember. Enter Docker Compose. With Docker Compose, we define all the necessary container runtime options in a YAML file and then use a simple command to run everything. Docker Compose can easily build, run, and turn off containers; attach volumes; inject environment variables; and more. Docker Compose is the first and easiest container orchestration tool we'll use. Working with Docker Compose is an important step on our way to using Kubernetes for container orchestration. Let's get started automating our container builds and pushes with GitHub Actions.

6.1 Building and pushing containers with GitHub Actions

Building container images for our applications should be as automated as possible; we have a few options to make this happen. First, we could use Docker Hub to automatically build images for us through a GitHub integration, or we could use GitHub Actions directly to build our images.

While it is great that Docker Hub can handle our automated builds, it causes a key problem that we want to avoid: another location for us to store any secrets, like keys or passwords. After we build a container, we want to be able to update the system(s) that use these container images. While this is possible with Docker Hub, it's far more challenging than just using GitHub Actions or the CI/CD pipeline next to where our code lives.

The workflow for GitHub Actions will introduce us to a new feature we haven't seen yet: *using a third-party GitHub Action*. Docker has an officially supported GitHub Action that helps ensure the `docker` command-line tool is available automatically in our GitHub Action Workflows. The tool is called `setup-buildx-action` and is very easy to implement

(see GitHub documentation at <https://github.com/docker/setup-buildx-action>). Before we do, let's talk about what third-party GitHub Action tools are.

6.1.1 Docker login and third-party GitHub Action tools

GitHub Actions is both a term for the CI/CD pipeline workflows as well as what I call plugins. We have already seen these plugins in previous workflows using `actions/checkout` and `actions/setup-python`. These are first-party GitHub Actions plugins because they are maintained by GitHub. There are many third-party plugins that you can use as well, which is what we'll implement in this section.

Keep in mind that using third-party GitHub Action plugins is *not* required, but it can help reduce the time it takes to run any given workflow. Third-party GitHub Action plugins also have an inherent risk because they can expose your workflows to untrusted parties.

Now we are going to use the Docker-maintained GitHub Actions plugin called `login_action` (see <https://github.com/marketplace/actions/docker-login>). This plugin simplifies the login process and allows us to use GitHub Actions secrets to hide our Docker Hub username and personal access token password.

Before we can use this plugin, we must create a Personal Access Token on Docker Hub and add it to our GitHub Actions secrets. Here's the step-by-step process:

- 1 Log into Docker Hub at <http://hub.docker.com/login>.
- 2 Navigate to your Account Security at <https://hub.docker.com/settings/security> or as follows:
 - Click your profile icon in the top right.
 - Click Account Settings in the drop-down menu.
 - Click Security on the left-hand side.
- 3 Click New Access Token:
 - Give your token a name (e.g., RK8s GitHub Actions).
 - Select Read & Write for the scope (we do not need Delete Permission).
 - Click Create.
 - Copy the token to add as GitHub Actions secrets for Python and JavaScript.
- 4 Navigate to each of your GitHub Action repositories (mine are <https://github.com/jmitchel3/roadtok8s-py> and <https://github.com/jmitchel3/roadtok8s-js>):
 - Navigate to Settings in the top-right.
 - Navigate to Secrets and Variables and click Actions on the left-hand side.
 - Click New Repository Secret.
 - Add `DOCKERHUB_USERNAME` as the name, with your username as the secret.
 - Add `DOCKERHUB_PASSWORD` as the name, with your newly created token as the secret.
 - Add `DOCKERHUB_REPO` as the name, with the value of the repository name you want to use. In my case, I'll have `rk8s-py` and `rk8s-js` as my values.

What these steps enable us to do is, first, log in to Docker Hub safely and securely and, second, tag our images based on secrets (i.e., the tag is not exposed) in a GitHub Actions workflow using `docker build -t ${ secrets.DOCKERHUB_USERNAME }/${ secrets.DOCKERHUB_REPO }:0.0.1` or whatever tag we choose. Let's build the workflow for both of our applications now (it's about the same for both).

6.1.2 A GitHub Actions workflow for building containers

Docker build and push commands are essentially the same for every project, with the variations being related to the way we tag the container image itself. This means that our GitHub Action workflow file can be re-used again and again and again. We will use the following tags to tag our images:

- `${ secrets.DOCKERHUB_USERNAME }/${ secrets.DOCKERHUB_REPO }:`
latest (as the default)
- `${ secrets.DOCKERHUB_USERNAME }/${ secrets.DOCKERHUB_REPO }:${ github.sha }`. `github.sha` is a built-in value that relates to the SHA of the specific commit which you can read about in the docs.

The following listing shows exactly how simple it can be to containerize an application this way.

Listing 6.1 Building containers with GitHub Actions

```
name: Build & Push Container
on:
  workflow_dispatch:

jobs:
  build-push:
    runs-on: ubuntu-latest
    env:
      DH_USER: ${ secrets.DOCKERHUB_USERNAME }
      REPO: ${ secrets.DOCKERHUB_REPO }
    steps:
      - uses: actions/checkout@v3
      - uses: docker/login-action@v2
        with:
          username: ${ secrets.DOCKERHUB_USERNAME }
          password: ${ secrets.DOCKERHUB_TOKEN }
      - name: Build the container
        run: |
          docker build -f Dockerfile \
            -t "$DH_USER/$REPO:latest" \
            -t "$DH_USER/$REPO:${ github.sha }" \
            .
      - name: Push the container images
        run: |
          docker push "$DH_USER/$REPO" --all-tags
```

For readability, we used the `env` variables for `DH_USER` and `REPO` to represent our store secret for `USER` and `REPO`, respectively. This means that `"$DH_USER/$REPO"` translates to `${{ secrets.DOCKERHUB_USERNAME }}/${{ secrets.DOCKERHUB_REPO }}`. and `codingforentrepreneurs/rk8s-js` depending on the repo.

Commit this file to your Git repo, push it to GitHub, and give it a whirl to see if you can successfully build both applications. If you run into errors, be sure to use the reference Python Repo (<https://github.com/jmitchel3/roadtok8s-py>) and Node.js Repo (<https://github.com/jmitchel3/roadtok8s-js>) for this book to compare your code to the working code. Now that we have our container images built and hosted on Docker Hub, let's start learning about another great Docker feature called Docker Compose.

6.2 Managing a container with Docker Compose

Docker Compose is our first step toward orchestrating our containers. To me, *orchestration* just means ensuring our containers run exactly how we intend them to run. This might include building, it might include different environment variables, and it might just include the container's repo. As we get more sophisticated with orchestration, we will learn that it also means *ensuring the health of a container*, but for now, we'll keep our orchestration simple: *building* and *running* containers.

Here are two of the simplest Docker commands you should memorize:

- `docker build -f Dockerfile -t your-tag .`
- `docker run -p 4321:4321 -e PORT=4321 your-tag`

If you are having trouble with these two commands, please review chapter 5. While knowing *how* these commands work should be memorized, the *parameters* to these commands do not need to be. In other words, each project is different, and you'll likely have different parameters for each one. This is where Docker Compose comes in.

Docker Compose will help us remember the parameters we need to build and run our containers correctly. Even if you never use Docker Compose directly, the Docker Compose configuration file, known as `compose.yaml` (also `docker-compose.yaml`), can be a great place to store all the parameters you need to build and run your containers.

6.2.1 Why we need Docker Compose

Do you remember which port you want your Python container to run on? How about the Node.js one? Is it 4321? Is it something else? What about the environment variables? Do you remember which ones you need to set? Using Docker means we abstract away a number of runtime elements, which you would need to know to answer these questions.

You will also come to find that your `docker run` commands are getting longer and longer, such that

```
docker run -v path/to/local/vol-b:/path/to/container/vol-b -e ENV_
  VAR=VALUE -e PORT=9090 -e DEBUG=1 -env-file ./path/to/env/.dev-file -p
  8000:8000 -d my-container-image-b
```

might actually be the configuration you need to run your application. Now try to do the same thing for three other container types, and you'll find it's a nightmare to remember.

Enter Docker Compose. Docker Compose can automate all of the commands we need to build and run our containers with just one: `docker compose up --build`. Before we explore the commands to use Docker Compose, let's review a sample configuration file known as `compose.yaml` or `docker-compose.yaml` (both filenames are supported). The following listing is a sample that might be used with a web application with an existing Dockerfile.

Listing 6.2 Your First Docker Compose file

```
services:
  web:
    build: .
    ports:
      - "3001:3001"
```

This code translates to

```
docker build --tag <folder-dirname>-web .
docker run --publish 3001:3001 <folder-dirname>-web
```

Do these look familiar to you? I should hope so. This `compose.yaml` file is just scratching the surface at this point, and it's even missing the declaration that replaces the `-f Dockerfile` argument we used previously. The point of seeing this now is to help us get a glimpse into what Docker Compose can do for us. Now that we have a taste of Docker Compose, let's put it into practice with our Python application.

6.2.2 Docker Compose for our Python app

Our Python application is more than ready to use Docker Compose to build our container because we have a working application, a Dockerfile, and environment variables. To build and run this container, we need to do the following:

- 1 Navigate to the root of our project with `cd ~/dev/roadtok8s/py`.
- 2 Run our build command with `docker build -f Dockerfile -t rk8s-py`.
- 3 Run our container with `docker run -e PORT=8080 -p 8080:8080 rk8s-py`

This process is nearly identical to Docker Compose, with a key aspect missing: runtime arguments. At this stage of using Docker Compose, we will focus on removing *all* the runtime arguments. To do this, let's first understand the key elements of a configuration for Docker Compose:

- **services**—This block will define any containerized applications we want to manage with Docker Compose. They are called services because Docker Compose can manage multiple containers at once while providing a network between them.
- **web**—We will give our first containerized application the service name of web. The service name is slightly different than the image name and tag name, but not by much. Service names will become clearer as we move further along in this chapter.
- **build:context**—The path to the location we want to run our build command from. We could use a different directory if needed or use the current directory (e.g., the one next to `compose.yaml`).
- **build:dockerfile**—The path to the Dockerfile we want to use relative to the `build:context` path.
- **ports**—The same as the `-p` argument we used in our `docker run` command.
- **environment**—The same as the `-e` argument we used in our `docker run` command.

Docker Compose has a lot more configuration options that are available to us, but these are among the most common and, in our case, the only ones necessary for our Python application.

Navigate to the root of your Python project (`~/dev/roadtok8s/py`) and create `compose.yaml` with the contents in the following listing.

Listing 6.3 Docker Compose for our Python app

```
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - PORT=8080
```

To build, run, and stop our container, we will use the following commands:

- `docker compose build` will build our container.
- `docker compose up` will run our container.
- `docker compose down` will stop our container.

Give these commands a try. Figure 6.1 shows a clipping of the build command (`docker compose build`).


```

% docker compose build
[+] Building 6.9s (16/16) FINISHED
=> [web internal] load .dockerignore 0.0s
=> => transferring context: 92B 0.0s
=> [web internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 798B 0.0s
=> [web internal] load metadata for docker.io/library/python:3.11.4 2.1s
=> [web auth] library/python:pull token for registry-1.docker.io 0.0s
=> [web internal] load build context 0.0s

```

Figure 6.1 Building a Python container with Docker Compose

Figure 6.2 shows a clipping of the run command (`docker compose up`).

```

% docker compose up
[+] Running 2/0
✓ Network roadtok8s-py_default Cre... 0.0s
✓ Container roadtok8s-py-web-1 Cre... 0.0s
Attaching to roadtok8s-py-web-1
roadtok8s-py-web-1 | [2023-08-04 00:11:49 +0000] [7] [INFO] Starting gunico
rn 21.2.0
roadtok8s-py-web-1 | [2023-08-04 00:11:49 +0000] [7] [INFO] Listening at: h
ttp://0.0.0.0:8080 (7)

```

Figure 6.2 Running a Python container with Docker Compose

With the Docker Compose running (after using `docker compose up`), open a new terminal window, navigate to your project, and run `docker compose down` to stop the container. Figure 6.4 shows a clipping of the stop command (`docker compose down`).

```

% docker compose down
[+] Running 2/2
✓ Container roadtok8s-py-web-1 Re... 10.1s
✓ Network roadtok8s-py_default Re... 0.1s
%

```

Figure 6.3 Stopping a Python container with Docker Compose

Docker Compose can be stopped in two ways:

- Using Control + C, like many other processes.
- Using `docker compose down` from another terminal window. This command should be your go-to command for stopping containers, as it's managed by Docker Compose and does a better job of cleaning up after itself. We'll revisit this command in the future.

Docker Compose missing?

If you're on a Linux machine and Docker Compose is not installed, consider using the following commands:

```
sudo apt-get update
sudo apt-get install docker-compose-plugin
```

If that fails, you may have to use an older version of Docker Compose. You can install this version via the Docker Compose Python package (<https://pypi.org/project/docker-compose/>) with `python3 -m pip install docker-compose --upgrade`.

Once that's installed, you will use `docker-compose` (notice the use of a dash -) instead of `docker compose` whenever you need to use Docker Compose.

This Python-based version of Docker Compose may require additional changes to your configuration. Refer to the official Docker Compose documentation (<https://docs.docker.com/compose/install/>) for more information.

Now that we have our first look at Docker Compose with our Python application, let's shift gears for our Node.js application; spoiler alert: it's going to be almost identical.

6.2.3 Docker Compose for our Node.js app

At this point, I would challenge you to create your own `compose.yaml` file for your Node.js application because it is almost identical to what we have already seen. As a quick recap, here are the steps we need to take to build and run our Node.js container:

- 1 Navigate to the root of our project with `cd ~/dev/roadtok8s/js`.
- 2 Run our build command with `docker build -f Dockerfile -t rk8s-js`.
- 3 Run our container with `docker run -p 3001:3000 -e PORT=3001 rk8s-js`.

The build-and-run process, aside from various arguments, is identical no matter what container you are using. This is the beauty of Docker Compose; it puts those arguments into a format that's easily repeated and very portable. Now navigate to the root of your Node.js project (e.g., `~/dev/roadtok8s/js`) and create `compose.yaml` with the contents from the following listing.

Listing 6.4 Docker Compose for our Node.js app

```
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - PORT=3000
```

Once again, we can build, run, and stop our container, respectively, with the following:

- `docker compose build`
- `docker compose`
- `docker compose down`

I will let you explore this on your own project to verify it's working correctly. Now that we understand the fundamentals of managing containers with Docker Compose, let's start to explore other kinds of apps your projects might need: *stateful* apps.

6.3 Stateful containers with volumes

Containers can be easily built and destroyed; they are designed to be ephemeral. Whenever you think about files *inside* a pure container, you should think of *temporary*. But what if you want to have persistent changes occur within a container? Enter volumes.

The runtime of a container is stateless. To make it stateful, you add a volume. A *volume* is basically a folder from your local machine, or operating system, which is mounted into the container at run time. The volume will persist even if the container is destroyed, rebuilt, stopped, started, and so on.

The best way to get familiar with volumes is by using them so we can see just how simple they truly are. We'll start by using volumes with just pure Docker commands.

6.3.1 Using volumes with Docker

To use a volume with the Docker CLI, we will include the `-v` flag along with the argument for `<local-folder>:<container-folder>`. In this case, `<local-folder>` is a placeholder for the path to a folder on our local machine we want to attach to our container. `<container-folder>` is a placeholder for the path we want to *mount* this folder in our container. Here's a couple examples:

- `docker run -v ~/dev/roadtok8s/py/data:/app/data/...`—This will mount the folder `~/dev/roadtok8s/py/data/` into the container at `/app/data/`. The `~/dev/roadtok8s/py/data/` will be the same anywhere on the machine.
- `docker run -v data:/app/data/ ..`—This will mount the relatively located folder `data` into the container at `/app/data/`. This data folder will be relative to the folder the command is run (e.g., using `docker run -v data:... in path/to/some/folder` will assume that `data` exists at `path/to/some/folder/data`).

You can attach a volume to any container. Detaching the volume is as simple as stopping the container and running it again without the `-v` flag.

Before we use volumes, let's recap the various arguments our `docker run` command needs:

- `docker run` is the command to run a container.
- `-v` is the flag for attaching a volume.

- `--name` is the flag for naming our running container (this makes it easier to stop or enter the shell).
- `<image-name>` is the final argument we'll use to specify our previously built image (e.g., with `docker build -t <image-name> -f Dockerfile`).

Using the `--name <my-custom-name>` flag allows us to run commands like `docker stop <my-custom-name>` to stop a container and `docker exec -it my-custom-name /bin/bash` to enter the bash shell. We'll do both in a moment. The following listing shows how we can use a volume with our Python application.

Listing 6.5 Using volumes with Docker

```
docker run -v ~/dev/roadtok8s/py/data:/app/data/ --name my-rk8s-py rk8s-py
```

The name of this runtime is `my-rk8s-py`, while the image name is `rk8s-py`. These two names tripped me up for a long time, so it's important for you to understand the difference: using `--name` just gives an arbitrary name to your container runtime, while the `<image-name>` must be a valid image. In other words, the runtime name is a variable; the image name is fixed.

With this container running, let's test our volume with the following commands:

```
echo "hello world" >> ~/dev/roadtok8s/py/data/sample.txt
docker exec my-rk8s-py cat /app/data/sample.txt
```

If done correctly, you should see the output of `hello world` from the `cat` command. This means that the volume is working correctly.

Let's log a view dates via the `date` command to a file and see if we can read them back. Run the following commands:

```
date >> ~/dev/roadtok8s/py/data/sample.txt
date >> ~/dev/roadtok8s/py/data/sample.txt
date >> ~/dev/roadtok8s/py/data/sample.txt
docker exec my-rk8s-py cat /app/data/sample.txt
```

NOTE The `date` command outputs the current date and time in the terminal. You can use any command you want here, but `date` is a simple one to use and track how this all works.

If done correctly, you should see the output shown in figure 6.4.

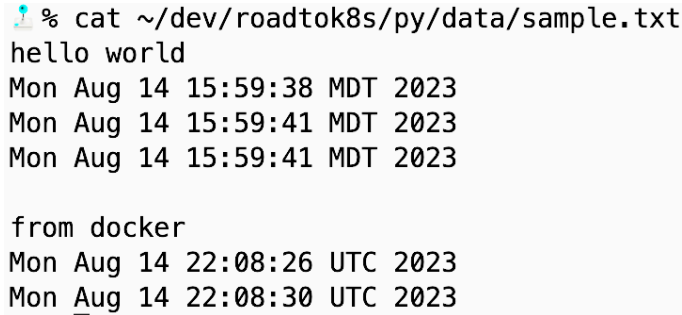
```
% docker exec my-rk8s-py cat /app/data/sample.txt
hello world
Mon Aug 14 15:59:38 MDT 2023
Mon Aug 14 15:59:41 MDT 2023
Mon Aug 14 15:59:41 MDT 2023
```

Figure 6.4. Volume output samples with Docker and a local machine

We can also add content to the file from within the container. Run the following commands:

```
docker exec my-rk8s-py bash -c "echo \"\" >> /app/data/sample.txt"
docker exec my-rk8s-py bash -c "echo \"from docker\" >> /app/data/sample.txt"
docker exec my-rk8s-py bash -c "date >> /app/data/sample.txt"
docker exec my-rk8s-py bash -c "date >> /app/data/sample.txt"
cat ~/dev/roadtok8s/py/data/sample.txt
```

Figure 6.5 shows the output of these commands.



```
% cat ~/dev/roadtok8s/py/data/sample.txt
hello world
Mon Aug 14 15:59:38 MDT 2023
Mon Aug 14 15:59:41 MDT 2023
Mon Aug 14 15:59:41 MDT 2023

from docker
Mon Aug 14 22:08:26 UTC 2023
Mon Aug 14 22:08:30 UTC 2023
```

Figure 6.5 Volume output samples with Docker and a local machine

This simple example shows us a few key items about Docker volumes:

- Volumes are a way to persist data outside of a container.
- Volumes can be accessible from the container or the local machine.
- Changes to files happen instantly (e.g., the files are not being synced or copied like in a Docker build).
- Volumes are used at runtime.
- Although it's not always recommended due to potential file corruption or read and write problems, volumes can be used to share data between containers.

Now that we have a basic understanding of volumes, let's use them with Docker Compose.

6.3.2 Using volumes with Docker Compose

Continuing on the path of using our local volume located at `~/dev/roadtok8s/py/data/`, we'll now use Docker Compose to mount this volume into our container at the path `/app/data`. The following listing shows how we can use a volume with our Python application.

Listing 6.6 Docker Compose configuration with volumes

```
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - PORT=8080
    volumes:
      - ~/dev/roadtok8s/py/data:/app/data
```

Now we can run the following:

- `docker compose up`—To bring up our web application docker compose service. Be sure to stop any other containers running at PORT 8080.
- `docker compose exec web ls /app/data`—To list out all of the files in our volume mounted on our container.
- `docker compose exec web bash -c "date >> /app/data/sample.txt"`—To append a new date to our sample.
- `docker compose exec web cat /app/data/sample.txt`—To see the output of our sample file.

As we see with these commands, Docker Compose has shortcut our ability to use volumes with our containers. We can now use the `docker compose exec` command to run commands inside of our container instead of needing to know the `--name` of a container or for the more advanced Docker usage the container ID.

To stop our Docker Compose services and attempt to remove volumes we can run `docker compose down -v` or `docker compose down --volumes`. The reason this is an attempt is because Docker Compose intelligently knows that the volume `~/dev/roadtok8s/py/dev` is not managed by Docker Compose. This means that Docker Compose will not attempt to remove this volume.

Let's allow Docker Compose to manage volumes on our behalf. Doing this gives Docker Compose more power over the environment but also provides us with simplicity when it comes to managing the dependencies for our containers.

6.3.3 Built-in volume management with Docker Compose

Docker Compose has a top-level section called `volumes` where we can declare various volumes that we want Docker Compose to manage. Naming the declared volume is up to us (much like with services), so I'll use painfully obvious names to make it easy to distinguish.

Listing 6.7 gives us an example of a `compose.yaml` where we name a volume `mywebdata` and map that volume to `/app/data` on our container. In this case, Docker

Compose will manage `mywebdata` because it is declared in the `volumes` section; it does not exist within my local project and never will. While there is a way to access this volume directly on our host machine, it is unnecessary for us to do, and thus allowing Docker Compose to manage `mywebdata` is the best option in this case. What's more, this volume will be removed when we run `docker compose down -v` because it is managed by Docker Compose; your self-managed directory will not be removed with this command.

Listing 6.7 Managing volumes with Docker Compose

```
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - PORT=8080
    volumes:
      - mywebdata:/app/data

volumes:
  mywebdata:
```

Now if we run `docker compose up` and `docker compose down -v`, we will see the output in figure 6.6.

```
% docker compose down -v
[+] Running 3/3
✔ Container roadtok8s-py-web-1   Removed    10.2s
✔ Volume roadtok8s-py_mywebdata  Removed    0.0s
✔ Network roadtok8s-py_default   Removed    0.0s
```

Figure 6.6 Managing volumes with Docker Compose

The figure shows us that Docker Compose will remove a volume that *destroys* all files and data within that volume. I showed you the other way first because destroying volumes that should not be destroyed is terrible and something that we want to avoid.

The final way we'll discuss managing containers is by declaring an external volume within the `compose.yaml` directly. External volumes are not managed by Docker Compose and thus will not be deleted when `docker compose down -v` is run, making them ideal candidates for data we need to persist regardless of Docker or Docker Compose's status. Using the `external: true` configuration allows you to use a folder relative to `compose.yaml` as an external volume that Docker Compose will not attempt to manage. The following listing shows us how to do this.

Listing 6.8 Managing volumes with Docker Compose

```
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - PORT=8080
    volumes:
      - mywebdata:/app/data
      - myprojectdata:/app/other-data

volumes:
  mywebdata:
  myprojectdata:
    external: true
```

If you run `docker compose up` without creating the folder `myprojectdata`, you will get an error stating external volume "myprojectdata" not found. This is a good indication that Docker Compose is not managing this folder for you, and you must make it.

Another key aspect to note here is you can mix and match the various volume types as well as have many different volumes attached to your container. This means you can have a volume that is managed by Docker Compose, a volume that is managed by Docker Compose but is external, and a volume that is not managed by Docker Compose at all. If the data in the volume is important and needs to persist, I recommend not allowing Docker Compose to manage it but instead attaching directories you manage yourself. Letting Docker Compose manage a volume means that it can be destroyed at any time, even by accident via `docker compose down -v`; this is not something you want to happen to your critical data.

Now that we have a grasp of how to use volumes, let's take a look at a practical reason to use them: *databases*. To use databases effectively with Docker Compose, we have to understand one more key concept: *networking*.

6.4 Networking fundamentals in Docker Compose

Docker Compose is our first step into multiple container orchestration, and understanding how containers communicate with each other is a critical step. Up until this point, we have been using our localhost to access our Python or Node applications, regardless of what port we use and whether the app is using a container or Docker Compose. The localhost is often mapped to the IP address 127.0.0.1, which is a special IP address that always points to the current machine. Using localhost allows us to communicate with locally hosted applications.

Docker Compose allows us to modify this by providing a *network* for our containers to communicate with each other. This means that we can use the service name instead of localhost, or 127.0.0.1, to access our applications. Docker Compose automatically creates this network for us, but we can also create our own networks if needed.

In this section, we'll look at a few fundamentals of networking with Docker Compose as they pertain to container-to-container communication. Using this kind of communication is how we can use stateful containerized applications like databases to connect directly to stateless containerized applications like our Python or Node.js applications. Once we understand these fundamentals, we will have laid the foundation for what's to come with Kubernetes.

6.4.1 Container-to-container communication simplified

With Docker Compose, containers communicate with one another using the *service* name. This service name becomes our *hostname*. This means that `http://localhost:8080` becomes `http://web:8080` where *web* is the service name.

In our Python application's `compose.yaml` we'll add a new service that runs the NGINX public container image. This container image is simple to run in Docker Compose and, if called correctly, will return standard HTML content by default.

Listing 6.9 shows us how to add this new service to our `compose.yaml` file. We'll use the service name `my-nginx-service` for the NGINX image, just to help distinguish it from our `web` service.

Listing 6.9 NGINX service in Docker Compose

```
services:
  my-nginx-service:
    image: nginx
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - PORT=8080
    volumes:
      - mywebdata:/app/data

volumes:
  mywebdata:
```

Now that we have this new service, perform the following steps:

- 1 If the containers are running, run `docker compose down`.
- 2 Run `docker compose up`. This will download the NGINX container image if it's not available locally. It will also create our network named `roadtok8s-py_default`, which both services will use by default.
- 3 Run `docker compose exec web /bin/bash`. This will enter the bash shell for the web service.
- 4 Within the web service, run the command `curl http://my-nginx-service` or `curl http://my-nginx-service:80`.

After running the `curl` command, we will see the HTML that results from the standard NGINX page, as seen in figure 6.7.

```
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully in
stalled and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
root@68a2944a8c33:/app#
```



Figure 6.7 NGINX service response in Docker Compose

Over at our `docker compose up` terminal window, we will see that the `my-nginx-service` will indicate the request occurred, as seen in figure 6.8.

```
#1: start worker process 31
roadtok8s-py-my-nginx-service-1 | 2023/08/15 18:30:41 [notice] 1
#1: start worker process 32
roadtok8s-py-my-nginx-service-1 | 2023/08/15 18:30:41 [notice] 1
#1: start worker process 33
roadtok8s-py-web-1 | [2023-08-15 18:30:41 +0000] [7
] [INFO] Starting gunicorn 21.2.0
roadtok8s-py-web-1 | [2023-08-15 18:30:41 +0000] [7
] [INFO] Listening at: http://0.0.0.0:8080 (7)
roadtok8s-py-web-1 | [2023-08-15 18:30:41 +0000] [7
] [INFO] Using worker: uvicorn.workers.UvicornWorker
roadtok8s-py-web-1 | [2023-08-15 18:30:41 +0000] [8
] [INFO] Booting worker with pid: 8
roadtok8s-py-web-1 | [2023-08-15 18:30:41 +0000] [8
] [INFO] Started server process [8]
roadtok8s-py-web-1 | [2023-08-15 18:30:41 +0000] [8
] [INFO] Waiting for application startup.
roadtok8s-py-web-1 | [2023-08-15 18:30:41 +0000] [8
] [INFO] Application startup complete.
roadtok8s-py-my-nginx-service-1 | 172.30.0.3 - - [15/Aug/2023:18
:31:03 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.88.1" "-"
```

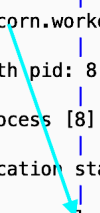


Figure 6.8 NGINX service response in Docker Compose

As we can see, the `web` service was able to communicate with the `my-nginx-service` service using the service name as the hostname. We can also see that our `compose.yml` file never exposes the `my-nginx-service` service to the host machine (our local machine) through a port. This means that the `my-nginx-service` service is only accessible from within the Docker Compose network. A quick way to get practice with this concept is to do the following:

- Run `docker compose down` to take the services and networks down.
- Change the service names in `compose.yml` to something else.
- Run `docker compose up` to bring the services and networks back up.
- Remove any and all declared ports.
- Attempt to run `curl` from within a bash shell (`docker compose exec my-service /bin/bash`) and see if you can access the other service.

Using the NGINX container made it easy for us since the default port is 80. Some containerized applications, like our Python and Node.js apps, do *not* use port 80 by default. It's important that when you perform communication within the network, you know what the correct port is.

This type of communication is great because it allows our Docker Compose services to use common ports *without* conflicting with ports on the host machine. While this is a nice feature, it can cause confusion while you are learning. So, when in doubt, add the `ports` arguments to your `compose.yml` file to expose the ports to your host machine. Now we are going to combine our knowledge of volumes and networking to create a database container that our Python application can use.

6.4.2 Databases and Docker Compose

When we use container-based database applications, we need to be aware of a few things:

- When we mount volumes, our data will persist. If we don't mount a volume, the data won't persist.
- Databases have standard ports that we need to be aware of. If you have too many projects on your host machine attempting to use the same port, you will run into a lot of unnecessary errors.
- `docker compose down` will persist data in Docker Compose-managed mounted volumes, while `docker compose down -v` will destroy data in those same volumes. This means your database data can be completely wiped just by using the `-v` flag with certain volume mounting types.
- Using the service name instead of the hostname is how your other services can connect.

With these items in mind, we will install *Redis* for our Python Application to use. Redis is among the most useful key-value data stores available, and it's incredibly easy to set up.

Before we add our Redis service to our `compose.yaml` file, we need to update our Python application's Dockerfile to install *redis-tools* so we can use the Redis command line client. The following listing shows us how to do this.

Listing 6.10 Updating our Python Dockerfile for Redis tools

```
...
# Run os-level updates
RUN apt-get update && \
    apt-get install -y python3-venv python3-dev python3-pip

# Install redis-tools
RUN apt-get install -y redis-tools
...
```

It's important to note that most open-source databases require additional system-wide drivers for client applications (e.g., in Docker) and language-specific packages (e.g., `requirements.txt` or `package.json`) to fully connect. This is true for containerized applications and non-container but still Linux-installed applications. In other words, if you want to use PostgreSQL or MySQL there are additional steps that are outside the scope of this book, but the process is nearly the same. With our Dockerfile updated, let's add a new service to our `compose.yaml` called `my-redis-service`, as in the following listing.

Listing 6.11 Redis service in Docker Compose

```
services:
  my-nginx-service:
    image: nginx
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - PORT=8080
    volumes:
      - mywebdata:/app/data
  my-redis-service:
    image: redis
    volumes:
      - myredisdata:/data

volumes:
  mywebdata:
  myredisdata:
```

Now we have all the conditions needed to update our web service to communicate with our Redis service. Follow these steps to see this in action:

- 1 Run `docker compose down --remove-orphans` to take the services and networks down. The `--remove-orphans` flag is useful if you ever change service names or remove services from your `compose.yaml` file.
- 2 Run `docker compose up --build`. In this case, the `--build` command is required because we changed our web service's Dockerfile.
- 3 Run `docker compose exec web /bin/bash` to enter the bash shell for the web service.
- 4 Within the bash session, run `redis-cli ping`. You should see `Could not connect to Redis at 127.0.0.1:6379: Connection refused`.
- 5 Now run `redis-cli -h my-redis-service ping`. You should see `PONG` as the response.

Aside from adjusting the environment, you will see once again that the hostname for Docker Compose services matches directly to what you declare in `compose.yaml`.

We can now iterate on our Python code to use Redis as a data store. We can also use the `redis-cli` to interact with our Redis service. While this is nice, the important bit is we now know how to do cross-container communication within Docker Compose using stateful and stateless containers that either have or do not have volumes.

The last key step for improving our containers is to use an external environment variable file called `dotenv`. This external file can be used with both Docker and Docker Compose; let's look at both of them.

6.4.3 *Environment variables with dotenv in Docker and Docker Compose*

Throughout this book, we have seen the `PORT` value pop up all over the place and typically as an environment variable. Changing this value is a simple yet effective way to highlight how much environment variables can affect our applications (containers or otherwise).

When it comes to hard-coding any type of application variable, we have to decide if it matters to the run time of the application itself. `PORT` is a good example of a run-time environment variable that can be changed while affecting how the application is accessed but not the logic of how it runs. In our case, we designed `PORT` to have this effect to adjust how it's accessed, but it has no effect on the application itself.

For a large number of environment variables, we need to ensure they are abstracted away from the application code. For example, passwords and API keys should never be written directly in the code itself. The obvious reason is that we commit code to a repository that can leak this sensitive data, thus causing a massive security vulnerability even if the repo is private. To help mitigate this, we can start leveraging `dotenv` (`.env`) files that allow us to abstract out our sensitive data into a file that will not be committed to a git repository.

dotenv files are used throughout the development process and, in some cases, are used in production systems. I tend not to recommend using dotenv in production, but doing so is certainly better than having sensitive data in your code. For production, I recommend tools like HashiCorp Vault or Google Secret Manager to manage sensitive data. Configuring both of these tools is outside the scope of this book.

To use dotenv files, we use the following syntax:

```
KEY=value
KEY="value"
KEY='value'
```

You can read more on this syntax at the official Docker reference (<https://docs.docker.com/compose/environment-variables/env-file/>).

We want to create a dotenv file for the `REDIS_HOST` variable so that our `redis-cli -h my-redis-service ping` can become `redis-cli ping`. A `REDIS_HOST` variable is a good example of an environment variable you would not expose in your code.

In the root of your Python project (`~/dev/roadtok8s/py`), create a file named `.env`. It's important that you use the dot prefix in the file name. This file will be ignored by Git and will not be committed to your repository (the `.gitignore` files we use already have `.env` in them). The following listing shows us how to add the `REDIS_HOST` variable to our `.env` file along with a few other examples that we won't use.

Listing 6.12 Redis host environment variable

```
REDIS_HOST=my-redis-service
DEBUG=1
RK8S_API_KEY="this-is-fake"
```

To use this `.env` file with Docker, it's as simple as `docker run --env-file path/to/.env -p 8080:8080 rk8s-py` or from the root of our project `docker run --env-file .env -p 8080:8080 rk8s-py`. Do you think running this container outside of Docker Compose will be able to communicate with the `my-redis-service` within Docker Compose? Give it a try.

To use this dotenv file within our `compose.yaml`, we can update any service configuration, much like in the following listing.

Listing 6.13 Using dotenv with Docker Compose

```
services:
  ...
  web:
    ...
    env_file:
      - .env
    environment:
      - PORT=8080
    volumes:
      ...
  ...
```

With this new configuration, we can now run `docker compose up` and `docker compose exec web /bin/bash` to enter the bash shell for the web service. Within the bash shell, we can run `redis-cli ping` and see the response: `PONG`. In the short term, this configuration is more complicated. In the long term, this configuration is simpler, more flexible, more portable, and ultimately more secure.

As you might imagine, there are many other ways to customize how to use both Docker and Docker Compose. This book is not designed for all the ways to customize those tools. Instead, we now have a practical model of using Docker Compose locally for developing our containerized applications. Now it's time to take this knowledge and deploy applications using containers with Docker and Docker Compose.

Summary

- Building production-ready containers should happen in a CI/CD pipeline to automate the process and ensure consistency and quality.
- Docker Compose is an effective way to manage various parameters needed to run a single container.
- To make a container remember data, we mount volumes to it, giving it a stateful nature instead of the default ephemeral nature.
- Docker Compose helps reduce the complexity of managing multiple containers at once while providing a number of useful features for managing volumes and container-to-container communication with built-in networking.
- It is never a good idea to hard-code sensitive information; always use environment variables when possible.
- Dotenv (.env) files are a common format to inject various environment variables into a container runtime through the Docker CLI or directly in Docker Compose.
- Docker Compose can manage a container's lifecycle from build to run to stop and can manage volumes for you if you declare them in the `compose.yaml` or the `docker-compose.yaml` file.

Deploying containerized applications

This chapter covers

- Installing Docker on a virtual machine
- Building and deploying Docker images to a VM
- Using Docker Compose in production
- Implementing a CI/CD pipeline
- Discussing the limitations of Docker Compose

When we first deployed our applications, we had to install a number of system-level dependencies (e.g., Supervisor, Python, Node.js, etc.) and runtime-specific dependencies (e.g., `pip install`, `npm install`). This was true for our local system as well as our remote production virtual machines. These dependencies will always be needed by their very definition, but Docker helps us to manage these dependencies in a more effective way, especially in production.

In production, the primary system-level dependency we'll need is a container runtime instead of individual application runtimes and their related system-level dependencies. Put another way, we only need to install Docker Engine and Docker Compose on our production virtual machine; we'll never need to install Python, Node, Java, MySQL, PostgreSQL, Redis, Ruby, etc. This is the magic of containers: one runtime to rule them all.

Before we deploy our applications, let's explore using Docker in production by deploying a simple and public containerized application: NGINX. We'll start here as a proof of concept for deploying more complex applications such as our Python and Node.js applications.

7.1 Hello prod, it's Docker

Deploying containers is as simple as installing Docker and running the command `docker run <image-name>` or `docker-compose up` (yes, the - is intentional). We will start with a simple deployment of the NGINX container image to limit how much we need to configure up front, much like we did with the non-containerized deployment of NGINX. The first step is provisioning a new virtual machine and installing Docker.

7.1.1 Provisioning a new virtual machine

I recommend starting this process with a fresh virtual machine, and at this point, you should be very familiar with creating a new virtual machine. If you need a refresher, here is a recap from chapter 3:

- 1 Create an account on Akamai Connected Cloud at linode.com/rk8s (book readers receive a promotional credit as well).
- 2 Log in to Akamai Connected Cloud (formally Linode).
- 3 Click the drop-down Create > Linode.
- 4 Use the following settings:
 - Distribution—Ubuntu 22.04 LTS (or latest LTS)
 - Region—Dallas, TX (or the region nearest you)
 - Plan—Shared CPU > Linode 2 GB (or larger)
 - Label—rk8s-docker
 - Tags—<optional>
 - Root password—Set a good one
 - SSH keys—<recommended> (Review appendix D to learn how to create one as well as add it to Linode.)
 - All other settings—<optional>
- 5 Click Create Linode.
- 6 After a few minutes, you will see an IP address appear in the Linode Console (figure 7.1).

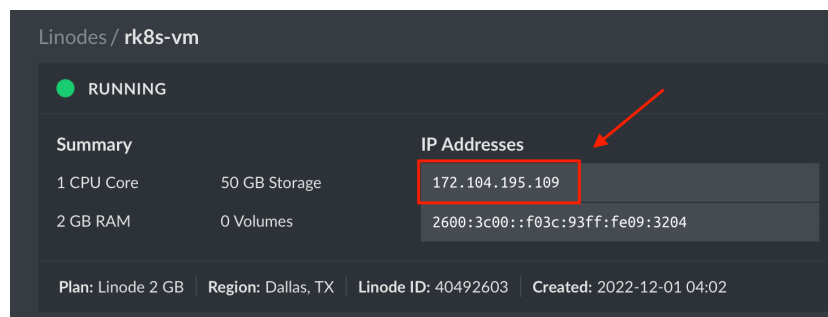


Figure 7.1 Linode virtual machine provisioned with an IP address

Once you have this virtual machine and IP address, verify that you can access it via SSH by running `ssh root@<ip-address>`. If you can connect, you are ready to install Docker.

7.1.2 Installing Docker on Ubuntu

When we install Docker on a Linux virtual machine, we install Docker Engine, which is the Docker runtime we will use. As a reminder, we installed Docker Desktop on our local machine. Docker Engine is not as heavy and does not have a GUI like Docker Desktop.

NOTE The term *Docker* means a lot of things, as we have come to learn. You may hear that Kubernetes no longer supports Docker, but that isn't true. Kubernetes uses a different runtime for running containers than Docker Engine. With additional configuration, Kubernetes can use Docker Engine as well as other runtimes, but it's not done by default. When Kubernetes moved away from Docker Engine, it caused confusion, as if Kubernetes no longer used Docker, which isn't true. Kubernetes uses containerd, which is an even lighter runtime than Docker Engine.

Like with our local system, there are a number of ways to install Docker on a virtual machine. My preferred way is by using the Docker installation script available at <https://get.docker.com>. If you want a more in-depth approach, review the official Docker Engine documentation (<https://docs.docker.com/engine/>). To install Docker Engine, be sure to start an SSH session on your virtual machine and then run the commands in the following listing.

Listing 7.1 Installing Docker Engine on Ubuntu

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

For those of you who are new to using these commands in Linux or Bash, here's how they break down:

- `Curl` is a way to download files from the internet via the command line.
- Using `-fsSL` with `Curl` is a combination of flags to follow redirects (`-f`), in silent mode (`-s`), suppress a progress bar (`-s`), and follow location-header redirects (`-L`). This flag ensures if any redirects happen, `Curl` will follow them and download the file without any command-line outputs.
- The command `sudo` ensures the proceeding command has full privileges to run this script. With many applications or installations, running it as the root user might not be recommended because it can have dramatic security implications. In this case, we are using a script from Docker that we trust.
- The command `sh` is how we can run this script as shell script. The command `bash` can be used as well.

After you run these commands, if you run `docker --version` you should see something resembling figure 7.2. The actual version that responds is not important as long as you see a version number and not a command not found error.

```
root@localhost:~# docker --version
Docker version 24.0.5, build ced0996
root@localhost:~# █
```

Figure 7.2 Docker Engine version

While this install script is great, it does not install Docker Compose, so let's do that now.

7.1.3 Installing Docker Compose on Ubuntu

Remember when I said that Docker Desktop is heavier than Docker Engine? One of the reasons is that Docker Engine does not include Docker Compose by default. To install Docker Compose, we'll use the built-in `apt` package manager, as shown in the following listing.

Listing 7.2 Installing Docker Compose on Ubuntu

```
sudo apt-get update
sudo apt-get install docker-compose-plugin
docker compose --version
```

NOTE Older versions of Docker Compose were installed via Python3 and used the command `docker-compose`. While the older version might work, I do not recommend it.

Now that we have both Docker and Docker Compose installed on our virtual machine, it's time to deploy our first containerized application directly with Docker. We are going to leave Docker Compose for a bit later.

7.1.4 Deploying NGINX

Deploying NGINX directly to Ubuntu is as easy as `sudo apt-get install nginx`. With Docker Engine, deploying NGINX with Docker is just as easy as running `docker run nginx`, but there's a catch. First, `docker run <container-image-name>` runs in the foreground by default. We can change this by using Docker's detached mode with `--detach` or `-d`, allowing the container to run in the background. This is similar to how we ran our Python and Node applications in the background with Supervisor. To run in NGINX in the background detach mode, use `docker run -d nginx`.

While continuously running containers in the background is a good idea, it brings up a new concern: how do we stop background running containers? We usually press Control + C, but that won't do anything here. What's more, we have no way to enter a Bash shell in this container.

Docker has a built-in process manager that you can access using `docker ps`, which will yield all of the running containers on your system. This works everywhere but is especially important now. Figure 7.3 shows the output of `docker ps` after running `docker run -d nginx`.

```
root@localhost:~# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
be52bea82b43	nginx	"/docker-entrypoint..."	5 seconds ago	Up 4 seconds	80/tcp	boring_jepsen

Figure 7.3 Docker process manager

This command gives us a few key pieces of information we need for any given container:

- **CONTAINER ID**—The container ID is a unique identifier for this container. This is how we can reference this container in other commands, as we will see shortly.
- **IMAGE**—The image name and tag used to create this container.
- **PORTS**—The ports that this container is currently using.
- **NAMES**—Docker will auto-assign names to containers unless you assign one.

We use the `CONTAINER ID` to do stop, view, and execute the bash shell:

- `docker stop <container-id>`—To stop a container
- `docker logs <container-id>`—To view the logs of a container
- `docker exec -it <container-id> bash`—To enter a bash shell in a container

When you run any Docker container image, it will automatically assign a name to the container unless we pass one with the flag `--name <your-container-name>` (e.g., the name `boring_jepsen` was autogenerated in this case). We can use this assigned name (e.g., the `NAME` columns of `docker ps` give us the name or names of our running containers) instead of the `CONTAINER_ID` to stop, review, and execute the `boring_jepsen` container:

- `docker stop <container-name>` OR `docker stop boring_jepsen`
- `docker logs <container-name>` OR `docker logs boring_jepsen`
- `docker exec -it <container-name> bash` OR `docker exec -it boring_jepsen bash`

If you are an observant reader, you will notice that there is a `PORT` associated with this container, and according to figure 7.3, the port is 80, which is the standard HTTP port. Now, you might be tempted to just visit `http://<ip-address>` in your browser, but that won't work. Do you remember why? The *internal* Docker port has not been exposed, or published, to the *external* or *system* port. Let's do that now.

Before we fix our running container, we should know about the command `docker ps -aq` because it will return a list of all running containers in a format that is easy to use with other commands. This is useful for stopping all running containers or restarting all running containers.

The final flag we need to discuss is the `--restart always` flag. This flag will ensure this application runs if Docker is restarted, the container fails (for some reason), or the virtual machine is restarted. This is a great flag to use for any container that you want to run continuously. Before we run the correct version of our NGINX container, let's stop all containers by running `docker stop $(docker ps -aq)`.

The following listing shows a complete example of running a background service, restarting the container if it fails, naming our container image, exposing a port, and running the NGINX container.

Listing 7.3 Stopping containers and publishing port

```
docker run -d --restart always --name my-nginx -p 80:80 nginx
```

If you need to manually stop and restart this running NGINX container, run `docker stop my-nginx`, `docker rm my-nginx` to remove the named container image and then repeat the command in listing 7.3.

You can now visit `http://<ip-address>` in your browser and see the NGINX welcome page. Congratulations, you have deployed your first containerized application in production! Now that we can deploy a production-ready application, let's discuss how to stage our applications using Docker and Docker Compose *before* we lock a production version.

7.2 Staging containers

While continuously deploying new versions of applications is a good idea, we very rarely go from development to production without staging environments. In other words, there will be a lag time between the production version of the application that is stable and the development version that is not stable (i.e., not yet ready for live traffic).

The point of this section is to understand techniques for staging containerized applications before they are deployed to production. We saw hints of doing so in chapter 5 when we ran our first container images and in chapter 6 when we started using Docker Compose, but now we'll solidify it with a number of purpose-built features of Docker.

The first and most obvious feature is how we *tag* images when we build them. Using specific tags is a quick and easy way to add versions to our code and to ensure we deploy the correct version. Tags are a nice way to signal a change, but they do not actually change our containers.

The process of preparing our application, containerized or not, has multiple stages. The first stage is the development stage, and it's typically on your local computer or a *development environment*. The second stage is the *staging* stage; you can consider this as a rehearsal for production that may need approval by other members of your team or just another review yourself. Staging an application is good practice because developers or team members intervene if something is technically correct (i.e., it passes all automated tests) but also *not* correct (i.e., it doesn't meet the requirements of the business or the team). The final stage is the *production* stage, which is the live version of the application that is available to the public or end users.

Staging environments often means changing internal container runtimes (e.g., Python 3.10 vs. Python 3.11), configurations (e.g., environment variables), the code itself (e.g., bug fixes, new features, etc.), and more. A few ways to do this are by using a different

- Dockerfile
- Docker Compose file
- Docker Compose file and a different Dockerfile
- Environment variables
- Entrypoint commands or scripts

Each is a valid way to modify how your application runs in or for different stages. Changing or using different Dockerfiles or Docker Compose files is one of the most common ways to prepare your application for production.

7.2.1 **Dockerfiles for multiple environments**

Docker can use different Dockerfiles during the build phase, and it's simple to do. The primary reason I specified the `-f Dockerfile` flag in all of the previous `docker build` examples was to hint at the simplicity of using different Dockerfiles.

For some projects, you might end up with multiple Dockerfiles for different environments or release phases such as

- Dockerfile for development
- `Dockerfile.prod` for production
- `Dockerfile.stage` for staging
- `Dockerfile.v2` for an unreleased version 2

To use this Dockerfiles, we might also tag them specifically for each phase or environment with

- `docker build -f Dockerfile -t my-container:dev .`
- `docker build -f Dockerfile.prod -t my-container:prod .`
- `docker build -f Dockerfile.stage -t my-container:stage .`
- `docker build -f Dockerfile.v2 -t my-container:v2 .`

When we are working in a development environment, or development stage, we can pick any tags we want for any Dockerfile; the flexibility is endless because the container image is likely just staying local. As soon as we want to *push* our container image anywhere, our tags must be valid. Container registries will host your image, and they require specific and valid tags to host them correctly. When we pushed to Docker Hub in chapter 6, we saw that we had to tag our images with three specific attributes: our Docker Hub username, a repo name, and a tag. Even self-hosted Docker registries require specific tagging to work correctly. For the vast majority of production systems, your container images *need* a valid container registry before they can be deployed.

How much you vary your build phases or stages will depend on the complexity of your applications, the requirements of the team or business, how many platforms you need to build for (e.g., ARM or x86), and how many staging environments you may need to support. For many smaller teams, you might only need two environments: development and production. Larger teams or applications often use many environments, many versions using semantic versioning (<https://semver.org/>), and many platforms.

The ability to use different Dockerfiles is the key takeaway as you start to move containers into production, while *tags* signify the change in the state of your containerized application. You will often have a different Dockerfile for your production stage and your development stage. Docker Compose can also help with staging multiple environments in a very familiar way.

7.2.2 Docker Compose for multiple environments

Docker Compose can also use the `-f <filepath>` flag to specify which Docker Compose file to use and thus which services to run. Each Docker Compose service is tied to a container, which can be tied to a Dockerfile. Once again, we can use this to our advantage to specify different Dockerfiles for staging different environments. Here are a few ways to stage different environments with Docker Compose:

- `docker compose -f compose.yaml up`
- `docker compose -f compose.prod.yaml up`
- `docker compose -f compose.stage.yaml up`
- `docker compose -f docker-compose.yaml up`

Do you notice anything missing? Docker Compose configurations are designed to abstract each argument any given container (or service) may need, including image tags (e.g., `my-repo/my-container:prod` or `my-repo/my-container:dev`).

Listing 7.4 contains a Docker Compose file from a previous chapter. In this compose. yaml file, I purposefully included the `dockerfile: Dockerfile` line to again hint at the fact we can easily change this as well.

Listing 7.4 Docker Compose with multiple Dockerfiles

```
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - PORT=8080
    volumes:
      - mywebdata:/app/data

volumes:
  mywebdata:
```

The ingredient we left out of this Docker Compose file is the ability to specify an image and a tag for that image. If the image or tags are unspecified, Docker Compose will set them for you based on the directory of the `compose.yaml` file. To update this, we specify the `tags:` within the `build` block and add an `image:` block. Copy the `compose.yaml` file and update it to `compose.prod.yaml` with the image and tag as seen in listing 7.5.

Listing 7.5 Tagging Docker Compose images

```
...
web:
  image: codingforentrepreneurs/rk8s-py:${TAG:-latest}
  build:
    context: .
    dockerfile: Dockerfile
    tags:
      - "codingforentrepreneurs/rk8s-py:${TAG:-latest}"
...
```

The `${TAG:-latest}` is a way to use the environment variable `TAG` to specify the tag of the image. If the `TAG` environment variable is not set, it will default to `latest`. This is a common pattern for specifying tags in Docker Compose files, and we have seen this syntax before with the `PORT` environment variable in our entrypoint scripts. To build a container image with Docker Compose, we can now run:

- `docker-compose -f compose.prod.yaml build`: This will tag the web image as `codingforentrepreneurs/rk8s-py:latest`.
- `TAG=v1.0.0 docker-compose -f compose.prod.yaml build`: This will tag the web image as `codingforentrepreneurs/rk8s-py:v1.0.0`.
- `TAG=v1.0.1 docker-compose -f compose.prod.yaml up --build`: This will tag the web image as `codingforentrepreneurs/rk8s-py:v1.0.1`.

Before you commit this to your GitHub repository, be sure to update your repository and replace `codingforentrepreneurs` with your Docker Hub username.

To ensure you have tagged your container images correctly, you can run `docker image ls <image-name>` or `docker image ls codingforentrepreneurs/rk8s-py` in our case. This will output something like figure 7.4.

```
👤 % docker image ls codingforentrepreneurs/rk8s-py
REPOSITORY                                TAG      IMAGE ID      CREATED      SIZE
codingforentrepreneurs/rk8s-py            latest   4dddf18b630e  2 hours ago  1.16GB
codingforentrepreneurs/rk8s-py            v1.0.0   4dddf18b630e  2 hours ago  1.16GB
```

Figure 7.4 Docker image list

Once you tag an image, you can push it to Docker Hub with `docker push <imagename> --all-tags`. Before you push all of your tags, you might want to remove a few. To remove image tags, you can run `docker rmi <image-name>:<tag>` or `docker rmi`

<image-id>. In my case and according to figure 7.4, that would be `docker rmi codingforentrepreneurs/rk8s-py:latest` or `docker rmi 4dddf18b630e`.

At this point, you should ensure that your local application directories (e.g., `~/dev/roadtok8s/py` and `~/dev/roadtok8s/js`) have the following Docker-related files in the root folder:

- `Dockerfile` and possibly `Dockerfile.prod`
- `compose.yaml` and `compose.prod.yaml`

If you do not have these files, make them now, commit them to your repository, and push them to GitHub. For a quick recap that will be

- 1 `git add Dockerfile*`: The `*` is a wildcard that will add both `Dockerfile` and `Dockerfile.prod`.
- 2 `git add compose*.yaml`: The `*` is a wildcard that will add both `compose.yaml` and `compose.prod.yaml`.
- 3 `git commit -m "Added Docker Support"`
- 4 `git push origin main`

Now that our Docker-ready applications are on GitHub, let's create a GitHub Action workflow to build, push, and deploy our containers to a production environment.

7.3 *GitHub Actions to deploy production containers*

In chapter 6, we created a GitHub Actions workflow to build and push our container into Docker Hub. In this section, we are now going to run a similar version of that workflow and modify it slightly to build, push, and deploy our containers into production.

Production containers are typically built on a different machine than the production host. This introduces a whole new caveat to the build-once, run-anywhere portability that containers offer. Different chips require different builds. Put another way, if you are using an ARM-based processor (e.g., Apple Silicon or Raspberry Silicon), when you run `docker build` it will, by default, create an ARM-based version of your container. This is great if you are running on an ARM-based machine but not so great if you are running on an x86-based machine. While ARM-based virtual machines are available, they are not nearly as common as x86-based virtual machines. We are using x86-based virtual machines in this book.

To solve this problem, we use GitHub Actions to build our containers. GitHub Actions workflows run on x86-based machines by default. This means that the containers we build will be x86-based and will run on our production virtual machines including when we use other container orchestration tools like Kubernetes. There is the option to do multi-platform builds, but that's outside the context of this book (see the official Docker documentation at <https://docs.docker.com/> to learn more about multi-platform builds).

Before you dive into this section, be sure to do the following:

- Create a new virtual machine on Akamai Connected Cloud (formally Linode) with Ubuntu 22.04 LTS, just like in section 7.1.1. If you are missing SSH keys, be sure to generate those as well (we have done this a number of times in this book).
- Add the newly created IP address to each application's GitHub Actions secrets as `AKAMAI_INSTANCE_IP_ADDRESS`. Again, we have done this a number of times throughout this book. Using a fresh virtual machine is important because we do not want a previously configured virtual machine to interfere with our new configuration.

The actual steps our GitHub Actions workflow is going to take are the following:

- 1 Check out our code so we can build an image.
- 2 Log in to Docker Hub.
- 3 Build our container with our Dockerfile.
- 4 Tag our container with relevant tags.
- 5 Push our container to Docker Hub.
- 6 SSH into our production virtual machine.
- 7 Verify Docker is installed; if not, install it. If Docker is not installed, also install Docker Compose.
- 8 Copy our `compose.prod.yaml` file to our production virtual machine.
- 9 Create a `dotenv` file (`.env`) and copy it to our production virtual machine.
- 10 Pull the latest container image with `docker compose pull`.
- 11 Run our container with `docker compose up -d`.
- 12 Clean up our SSH connection and `dotenv` file.

Each one of these steps will be translated into GitHub Actions workflow steps. With this in mind, let's get started on our workflow.

7.3.1 *Building and hosting a production container*

At this point, we should be well aware that a production container is the same as a nonproduction container from a technical perspective. The only difference is the environment in which it runs.

To build and host our production container, we are going to re-implement the GitHub Actions workflow from chapter 6. If you are comfortable with that workflow, this process will be the same, with a few minor changes. To get started, let's design our workflow to accomplish the following:

- Check out our code (essentially copy it).
- Log in to Docker Hub. Use the GitHub Actions secrets `DOCKERHUB_USERNAME` and `DOCKERHUB_TOKEN`.
- Build and tag our container. Use the GitHub Actions secret `DOCKERHUB_REPO` in combination with the GitHub Actions environment variable `GITHUB_SHA`.
- Push our tagged container image to Docker Hub.

As a reminder, your Docker Hub username (`DOCKERHUB_USERNAME`) needs to be stored in your GitHub repo's Action secrets. The Docker Hub token (`DOCKERHUB_TOKEN`) is a token you can create in your Docker Hub account settings. The Docker Hub repository (`DOCKERHUB_REPO`) is the name of your repository on Docker Hub; it can be the same as your GitHub repository name, but it does not have to be.

Verify that your GitHub Actions secrets values being used in your workflows (e.g., in `.github/workflows/`) are the same as what is in the Github Repo's stored Actions secrets and that they have not changed from what we did in chapter 6. While your values should not have changed, it's important that they remain the same as in chapter 6 to ensure our new workflow works as expected.

In one of your local applications (e.g., `~/dev/roadtok8s/py` or `~/dev/roadtok8s/js`), create a file `.github/workflow/build-push-deploy.yaml` starting with the contents of listing 7.6. You can work on both projects at the same time if you want, but I recommend sticking with one until it works every time.

Listing 7.6 GitHub Actions workflow to build and push containers, part 1

```
name: Compose - Build, Push & Deploy

on:
  workflow_dispatch:

jobs:
  deploy:
    runs-on: ubuntu-latest
```

As a reminder, the `runs-on` block shows us that we will be using `ubuntu-latest` to run this step's jobs, which will end up building our container image. Of course, we use Ubuntu on our production virtual machine as well.

Before we start declaring the steps, we can configure the job's environment variables with the `env:` declaration. As mentioned before, this can help make our workflow more concise and easier to maintain. The following listing shows the `env:` declarations we need for this job in this workflow.

Listing 7.7 GitHub Actions workflow to build and push containers, part 2

```
runs-on: ubuntu-latest
env:
  DH_USER: ${{ secrets.DOCKERHUB_USERNAME }}
  REPO: ${{ secrets.DOCKERHUB_REPO }}
  SSH_OPTS: '-o StrictHostKeyChecking=no'
  REMOTE: 'root@${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}'
  REMOTE_APP_DIR: '/opt/app'
```

The new additions to the `env` block are as follows:

- `SSH_OPTS`—When we use `ssh` and `scp` in an automated workflow, we may need to skip the host key check given the ephemeral and serverless nature of GitHub Actions. Using the flag `-o StrictHostKeyChecking=no` will skip this check.
- `REMOTE`—This is the remote user and IP address of our production virtual machine. We will use this to SSH or copy files (via `scp`) into our production virtual machine.
- `REMOTE_APP_DIR`—This is the destination directory for our Docker Compose file `compose.prod.yaml` and *nothing else*.

With these items in mind, let's configure the steps to build and push our container image to Docker Hub with the values in the following listing.

Listing 7.8 GitHub Actions workflow to build and push containers, part 3

```
...
jobs:
  deploy:
    ...
    env:
      ...
    steps:
      - name: Checkout code
        uses: actions/checkout@v3
      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${ secrets.DOCKERHUB_USERNAME }
          password: ${ secrets.DOCKERHUB_TOKEN }
      - name: Build the Docker image
        run: |
          docker build -f Dockerfile \
            -t "$DH_USER/$REPO:latest" \
            -t "$DH_USER/$REPO:${ github.sha }" \
            .
      - name: Push the Docker image to Docker Hub
        run: |
          docker push "$DH_USER/$REPO" --all-tags
```

With this code available, commit and push to your GitHub repository (e.g., `git add .` and `git push origin main`). After you do, confirm this workflow runs successfully in GitHub Actions. While it should run correctly based on our previous actions, we need to verify that before we move to the next step.

After you verify you can once again build and push container images to Docker Hub, let's start the process of configuring our production virtual machine.

7.3.2 Installing Docker and Docker Compose on a virtual machine with GitHub Actions

Earlier in this chapter, we saw how easy it is to install Docker and Docker Compose on a virtual machine with only a few commands. We are going to do the same thing here but with GitHub Actions.

You might be wondering if we should use Ansible to configure our virtual machine, just as we did in chapter 4. While that is a valid option and one you may consider adopting in the future, the simplicity of installing Docker with the shortcut script means Ansible is not necessary at this time. Here's what we'll do instead of using Ansible:

- `command -v docker`—This is a simple check to see if Docker is installed. If it is, we are done. If it is not, we need to install Docker.
- `curl -fsSL https://get.docker.com -o get-docker.sh` and `sudo sh get-docker.sh`—These are the same commands we used to install Docker in section 7.1.
- `sudo apt-get update` and `sudo apt-get install docker-compose-plugin`—This is the same command we used to install Docker Compose in section 7.2.

We can use these commands to create a condition statement that can be run with an SSH connection. The following listing configures our SSH key and shows how these commands are used.

Listing 7.9 Installing Docker on virtual machine with GitHub Actions workflow and SSH

```
...
jobs:
  deploy:
    ...
    env:
      ...
      SSH_OPTS: '-o StrictHostKeyChecking=no'
      REMOTE: 'root@${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}'
    steps:
      ....
      - name: Implement the Private SSH Key
        run: |
          mkdir -p ~/.ssh/
          echo "${{ secrets.SSH_PRIVATE_KEY }}" > ~/.ssh/id_rsa
          chmod 600 ~/.ssh/id_rsa
      - name: Ensure Docker and Docker Compose is installed on VM
        run: |
          ssh $SSH_OPTS $REMOTE << EOF
          if ! command -v docker &> /dev/null; then
            curl -fsSL https://get.docker.com -o get-docker.sh
            sudo sh get-docker.sh
            rm get-docker.sh
            # install Docker Compose
            sudo apt-get update
            sudo apt-get install docker-compose-plugin
          fi
        EOF
```

Now if you run this workflow, Docker and Docker Compose should be installed on your new virtual machine. Verify this with a new SSH session on your local machine with `ssh root@<ip-address> docker --version` and `ssh root@<ip-address> docker compose version`. Running these commands should result in an output like in figure 7.5.

```
% ssh root@172.232.161.201 docker --version
Docker version 24.0.5, build ced0996
% ssh root@172.232.161.201 docker compose version
Docker Compose version v2.20.2
```

Figure 7.5 Docker Compose version

NOTE If you have any SSH errors on your local machine, you might need to revisit how to install your SSH keys from section 4.1 or see appendix C for more information on SSH keys.

Now that we have Docker and Docker Compose installed on our virtual machine, run the workflow two more times. What you will see is this new installation step should run very quickly since Docker is already installed. You can also consider provisioning a new virtual machine to see how long it takes to install Docker and Docker Compose from scratch. Let's continue to the next step of our workflow, using Docker Compose to run our production application.

7.3.3 Using GitHub Actions to run Docker Compose in production

As we have already seen, Docker Compose makes using Docker much easier. The production version is also very easy to use. Here are a few conditions that we have already established to ensure smooth delivery of a production version of our containerized Docker-Compose-based applications:

- A production-ready Dockerfile
- A production-ready Docker Compose file (e.g., `compose.prod.yaml`)
- A production-ready container image
- A cloud-based production-ready host virtual machine
- Docker Hub hosted container images (either public or private)
- Docker and Docker Compose are installed on the host machine
- Production secret values stored on GitHub Actions secrets for our application repositories

Our GitHub Actions workflow already has a number of these conditions fulfilled, so let's start with using privately hosted Docker Hub container images. Private images take just one extra step to start using in production: logging in to Docker Hub via `docker login`. The following listing shows how to log in to Docker Hub with GitHub Actions and SSH.

Listing 7.10 Logging in to Docker Hub on a virtual machine using SSH

```
...
jobs:
  deploy:
    ...
    env:
      ...
      SSH_OPTS: '-o StrictHostKeyChecking=no'
      REMOTE: 'root@${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}'
    steps:
      ...
      - name: Ensure Docker and Docker Compose is installed on VM
        ...
      - name: Login to Docker Hub on VM
        run: |
          ssh $SSH_OPTS $REMOTE << EOF
            docker login -u $DH_USER -p ${{ secrets.DOCKERHUB_TOKEN }}
          EOF
```

The important part of this step is that you ensure that it is done through an SSH connection and not a GitHub Action like the `docker/login-action@v2` step.

Now let's add our environment variables to our `REMOTE_APP_DIR` location. The environment variables file, or `dotenv (.env)` file, will be created within the GitHub Actions workflow from stored GitHub Actions secret and then copied to our production virtual machine. After this file is created, we'll also copy `compose.prod.yaml` as `compose.yaml` to our production virtual machine to enable using `docker compose up -d` without specifying a compose file. The following listing shows how to do this.

Listing 7.11 Using SSH and SCP to add Dotenv secrets to a virtual machine

```
...
jobs:
  deploy:
    ...
    env:
      ...
      SSH_OPTS: '-o StrictHostKeyChecking=no'
      REMOTE: 'root@${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}'
      REMOTE_APP_DIR: '/opt/app'
    steps:
      ...
      - name: Create .env file
        run: |
          echo "MY_SECRET_KEY=${{ secrets.MY_SECRET_KEY }}" >> .env
          echo "API_ENDPOINT=${{ secrets.API_ENDPOINT }}" >> .env
      - name: Ensure remote directory exists
        run: |
          ssh $SSH_OPTS $REMOTE mkdir -p $REMOTE_APP_DIR
      - name: Copy GitHub Actions .env file to Remote VM
        run: |
          scp $SSH_OPTS .env $REMOTE:$REMOTE_APP_DIR/.env
```

The `.env` file is created right in the GitHub Actions workflow, so we can copy it to our virtual machine, and eventually, Docker Compose can use it when running `docker compose up`.

NOTE From a security perspective, it might be a good idea to encrypt and decrypt the `.env` file during this process, but doing so is outside the scope of this book. Securing secrets is a never-ending process, so you must stay vigilant in doing so. HashiCorp Vault and Google Secrets Manager are both valid options for runtime secret storage, but they may still require moving secrets from one place to a production virtual machine(s).

Keep in mind that it is necessary to create the app folder (e.g., the `REMOTE_APP_DIR`) on our machine before copying the `.env` file to that folder:

- `ssh $SSH_OPTS $REMOTE mkdir -p $REMOTE_APP_DIR` creates the folder on the remote VM.
- `scp $SSH_OPTS .env $REMOTE:$REMOTE_APP_DIR/.env` copies the `.env` file to the folder.
- `REMOTE_APP_DIR` is the folder definition we created in the `env:` block of our workflow.

Now that our `.env` has been created on our virtual machine, we need to bring our Docker Compose configuration file over. To ensure you and I have the same configuration, let's create the `compose.prod.yaml` files for each application (e.g., `~/dev/roadtok8s/py/compose.prod.yaml` and `~/dev/roadtok8s/js/compose.prod.yaml`).

First, we'll look at our Python application's `compose.prod.yaml` file. There are a few key attributes to notice in this file:

- `ports`—This is set to `80:8080`. Port 80 is the recommended port for exposing production Docker apps (e.g., `http://my-ip:80` is the same as `http://my-ip` as far as web browsers are concerned).
- `env_file`—This is set to `.env`, the file we created in the GitHub Actions workflow. This definition assumes a relative path, which means that `.env` and our Docker Compose configuration must be in the same directory.
- `build`—This is no longer included since the production compose file is only used for *running* images.
- `image`—This is set to `codingforentrepreneurs/rk8s-py:${TAG:-latest}`, which gives us the option to set the tag with the `TAG` environment variable on the virtual machine (e.g., not in `.env`). If the `TAG` environment variable is not set, it will default to `latest`. `image` can be public or private because our GitHub Actions Workflow will log in to Docker Hub before running `docker compose up -d` on our virtual machine.
- `restart: always`—This is set to ensure our container runs continuously and will restart if it fails for some reason.

Create `compose.prod.yaml` in the root of your Python application (e.g., `~/dev/roadtok8s/py/compose.prod.yaml`) with the contents in the following listing.

Listing 7.12 Docker Compose for production Python

```
services:
  web:
    restart: always
    image: codingforentrepreneurs/rk8s-py:${TAG:-latest}
    ports:
      - "80:8080"
    environment:
      - PORT=8080
    env_file:
      - .env
```

Repeat this process for our JavaScript Node.js application at `~/dev/roadtok8s/js/compose.prod.yaml`. Listing 7.13 shows the `compose.prod.yaml` file for our Express.js application. Notice it is to our Python application with a few minor changes.

Listing 7.13 Docker Compose for production Node.js

```
services:
  web:
    restart: always
    image: codingforentrepreneurs/rk8s-js:${TAG:-latest}
    ports:
      - "80:3000"
    environment:
      - PORT=3000
    env_file:
      - .env
```

The main difference is the default `PORT` value the container runs on; other than that, the files are nearly identical.

Now that we have our production Docker Compose configuration files, let's update our GitHub Actions workflow to copy this file, pull images, and run our containers. The keys to keep in mind are the following:

- Renaming to `compose.yaml` means we do not need to specify the file we will use.
- `compose.yaml` needs to exist next to `.env` on our virtual machine (e.g., `/opt/app/.env` and `opt/app/compose.yaml`).
- Running `docker compose pull` in our `REMOTE_APP_DIR` will pull the latest image(s) from Docker Hub based on our `compose.yaml` file.
- Running `docker compose up -d` in our `REMOTE_APP_DIR` will run our container(s) in the background at the correct ports.

The following listing shows how to do this for either application's `compose.prod.yaml` file.

Listing 7.14 Copying configuration, pulling Images, and running Docker Compose

```

...
jobs:
  deploy:
    ...
    env:
      ...
      SSH_OPTS: '-o StrictHostKeyChecking=no'
      REMOTE: 'root@${{ secrets.AKAMAI_INSTANCE_IP_ADDRESS }}'
      REMOTE_APP_DIR: '/opt/app'
    steps:
      ...
      - name: Copy compose.prod.yaml to VM
        run: |
          scp $SSH_OPTS compose.prod.yaml $REMOTE:$REMOTE_APP_DIR/compose.yaml
      - name: Pull updated images
        run: |
          ssh $SSH_OPTS $REMOTE << EOF
            cd $REMOTE_APP_DIR
            docker compose pull
          EOF
      - name: Run Docker Compose
        run: |
          ssh $SSH_OPTS $REMOTE << EOF
            cd $REMOTE_APP_DIR
            # run containers
            docker compose up -d
          EOF

```

Before we commit this file, it's often a good idea to clean up our SSH connection and remove the .env file from the GitHub Actions workflow. While doing so may not be strictly necessary as these workflows are destroyed once completed, I consider it a good idea to ensure the workflow limits any potential secret leak or security concerns. The following listing shows how to do this.

Listing 7.15 Cleaning up SSH and the dotenv file in GitHub Actions

```

...
jobs:
  deploy:
    ...
    env:
      ...
    steps:
      ...
      - name: Clean up .env file
        run: rm .env
      - name: Clean up SSH private key
        run: rm ~/.ssh/id_rsa

```

Be sure to add `MY_SECRET_KEY` and `API_ENDPOINT` as placeholder values in your GitHub Actions secrets on your GitHub repo(s). Any time you need to add or remove environment variables to your application, you will update the GitHub Actions workflow `.env` definition step, update GitHub Actions Secrets on your repo, push the changes, and run the workflow.

At this point, commit the workflow file (`git add .github/workflows/`) and the Docker Compose Configuration (`git add compose.prod.yaml`), push it (`git push origin main`) to your GitHub repository, and then run the GitHub Actions workflow. If you face any errors or problems, check the logs and the GitHub Actions secrets.

If there are no errors, congratulations! You have now deployed a production containerized application with Docker Compose and GitHub Actions! I suggest making modifications to your app now and seeing if the workflow is working correctly.

Getting to this point required a number of configuration items; the process is now repeatable and can be used for any number of applications. Before we continue to the next chapter, let's discuss some of the challenges with Docker Compose and the methods we implemented here.

7.4 The limitations of using Docker Compose for production

As your projects scale or get more complex, you'll learn that Docker Compose has some serious limitations. Here are a few I want to highlight:

- Scaling container instances—While technically you can add services, you can't easily scale multiple instances of the same running container.
- Health checks—Docker Compose does not have a built-in health check mechanism other than whether the container is running or not.
- Remote access—Docker Compose does not have a built-in way to remotely access a container's runtime (e.g., `docker compose run <service-name> bash` without SSH).
- Multi-virtual machine support—Docker Compose does not have a built-in way to run containers on multiple virtual machines by default.
- Ingress management—While technically possible with an NGINX service, there is no native way to manage incoming traffic with Docker Compose (e.g., `ip-address/api` goes to the `api` service, and `ip-address` goes to the `web` service).
- Public IP address provisioning—Docker Compose does not have a built-in way to provision a public IP address for a container or service. As we'll see in the next chapter, managed Kubernetes provides this ability. This means you can provision a public IP address for a container or service without needing to provision a virtual machine.

While there are a number of other limitations we won't discuss here, these might convince you to pursue leveraging Kubernetes, as we'll discuss in the next chapter. Before we do, let's review a sample of how you could scale containers with Docker Compose, regardless of whether it's in production or development.

7.4.1 **Scaling containers with Docker Compose**

Docker Compose does not have a built-in way to scale containers, so we'll use NGINX to give us an approach to how we can benefit from running the same container multiple times as duplicated Docker Compose services. NGINX has a number of features we have already seen in this book. One of those features is forwarding requests to different apps based on a path. As a reminder, we can forward the path `/about` to a Python application and the path `/api` to a Node.js application. This works because NGINX can forward requests on the local machine to different ports on that same machine (e.g., `localhost:8080` and `localhost:3000`).

It's important to remember that, on our localhost, we cannot run multiple applications on the same port. Ports are reserved for one process (or application) at a time. This is why we use different ports for different applications. But what if we use a different IP address instead of localhost? Can NGINX still forward the request regardless of the port? The answer is great news for us—yes it can! NGINX can easily forward requests to different virtual machines via IP addresses.

NGINX has one more trick up its sleeve: *load balancing*. We now understand that NGINX can forward requests to multiple destinations (e.g., `/about` and `/api`) while being served by different applications. NGINX only cares that the request destination is being served correctly or, in other words, that the app at `http://localhost:8080` returns a valid response to `/about`. NGINX does not care what application is serving the request. Given this, NGINX can forward requests to a list of multiple destinations, or in other words, we can provide NGINX with a number of valid destinations for a given path. This allows us to have, presumably, the *same app* running on `http://localhost:8080`, `http://localhost:8081`, `http://my-ip-address:80`, and `http://my-other-ip-address:80` and NGINX will rotate requests across these destinations. Rotating requests can be defined in a few different ways such as Round Robin (one after another), Weighted Round Robin (one preferred more than one after another), Least Connections (the least busy), IP Hash (based on the IP Address of the request), and several others. NGINX uses Round Robin by default but can be configured to use other methods (known as load balancing algorithms), although that is outside the scope of this book.

Listing 7.16 shows an basic example NGINX configuration file for implementing load balancing.

Listing 7.16 NGINX load balancing basics

```
upstream myapp {
    server localhost:8080;
    server localhost:8081;
    server my-ip-address:80;
    server my-other-ip-address:80;
}

server {
    listen 80;
    server_name localhost;
```

```

location / {
    proxy_pass http://myapp;
}

location /api {
    proxy_pass http://localhost:3000;
}
}

```

The upstream `<your-name>` definition is how we can specify the various destinations we want to use.

Now we can modify this configuration to work with Docker Compose services. While we are not going to implement this in our current applications, the following listings can serve as examples you can try on your own. Here are the steps we need to do to configure NGINX to load-balance across multiple Docker Compose services:

- 1 Define the `compose.yaml` with multiple service names. For this example, we'll use `bruce`, `tony`, and `thor` as our service names.
- 2 With the service names as our *hostnames* (that is, replacing the IP address and `localhost`), we'll configure our upstream definition in an NGINX configuration file.
- 3 Update the `compose.yaml` configuration with an NGINX service that uses the new NGINX configuration file.

Listing 7.17 shows a minimized Docker Compose file with only the application services listed without the `ports` definition. As a reminder, Docker Compose can do cross-service communication; thus no ports are needed to expose these services to the host machine.

Listing 7.17 Docker Compose with multiple services of the same container

```

services:
  bruce:
    image: codingforentrepreneurs/rk8s-py
    restart: always
  tony:
    image: codingforentrepreneurs/rk8s-py
    restart: always
  thor:
    image: codingforentrepreneurs/rk8s-py
    restart: always
  scott:
    image: codingforentrepreneurs/rk8s-js
    restart: always

```

With these services declared, let's look at a NGINX configuration file (e.g., `custom-nginx.conf`) that we could use to implement these different services, in listing 7.18.

Listing 7.18 NGINX upstream configuration

```
upstream myapp {
    server bruce:8080;
    server tony:8080;
    server thor:8080;
    server scott:3000;
}

server {
    listen 80;
    server_name localhost;

    location / {
        proxy_pass http://myapp;
    }
}
```

We now see `bruce:8080`, `tony:8080`, `thor:8080`, and `scott:3000` as our upstream destinations instead of `localhost` or an IP address. Docker Compose will automatically direct these requests to the correct service because the service names are the same as the host names. The ports `8080` and `3000` are the default ports defined in the container images for the different services referenced.

Finally, to update our Docker Compose configuration, we need to use our custom NGINX configuration file (e.g., `custom-nginx.conf`). Listing 7.19 shows how to do this.

Listing 7.19 Docker Compose with service load balancing in NGINX

```
services:
  nginx:
    image: nginx
    ports:
      - 8000:80
    volumes:
      - ./custom-nginx.conf:/etc/nginx/conf.d/default.conf
  bruce:
    ...
  tony:
    ...
  thor:
    ...
  scott:
    ...
```

With this configuration, you could run `docker compose up`, navigate to `localhost:8000`, and you should see NGINX load-balancing in action. If you reload the page enough times, you should see the different container instances being used. Modify this example as you see fit to better understand how this is functioning.

While using NGINX to load-balance services in Docker Compose is very nice to know, it still leaves us with a number of challenges that fall outside the scope of what Docker Compose can realistically do. In the next chapter, we'll look at how Kubernetes can help us solve many of the challenges with scaling and managing multiple containers.

Summary

- Installing Docker on virtual machines is generally easier than most programming languages and frameworks.
- Docker Compose is an effective way to run, manage, and update containers in development and production.
- Docker Compose managed volumes can potentially cause major issues, including data deletion, if not configured correctly.
- GitHub Actions provides an effective way to build production-ready containers that work on x86-based virtual machines by default.
- Docker Compose can manage a handful of containers and services, but it starts to fail as we scale.
- Load-balancing with NGINX is an effective way to scale applications, containerized or not, without needing to use more robust orchestration methods.
- Using documents, such as `compose.yaml` or a GitHub Actions workflow, to manage configurations is an effective way to ensure consistency and repeatability while providing a single source of truth for an application's configuration.

Managed Kubernetes Deployment

This chapter covers

- Provisioning a managed Kubernetes cluster
- Installing the Kubernetes CLI (`kubectl`)
- Connecting to a cluster with `kubectl`
- Running a container with a Deployment
- Exposing a deployment with a Service
- Container-to-container communication with DNS
- Load-balancing containers with ingress services

We have now reached our primary destination: the land of Kubernetes. Put simply, Kubernetes runs and manages containers across a cluster of virtual machines. Kubernetes can start, stop, and restart containers; run 0 to N number of container instances; switch or roll back versions; unlock container-to-container communication at scale; manage volumes and storage; inject environment variables; and so much more. K8s, a shorthand for Kubernetes (pronounced “kay eights”), also has a massive third-party ecosystem that extends its capabilities even further. The sheer number of features and third-party tools can make learning Kubernetes seem downright daunting. This chapter focuses on a few key features of Kubernetes that will accomplish two primary objectives: deploy our applications and build a practical foundation for learning more advanced Kubernetes topics.

Starting in chapter 5, we learned that containers open a whole new world of portability while also reducing the overhead in production-system configuration. This

reduction happens because with containerized applications our virtual machines mostly need just a container runtime (e.g., Docker Engine) to run our apps with very little other configuration. Put another way, using containers streamlines the deployment process by spending significantly less time configuring virtual machines for various runtimes (e.g., Python, Node.js, or Java) while also reducing the time needed for additional app-specific configuration on the virtual machine. While containerized applications still need app-specific configurations to run, this type of app-specific configuration can occur long before the app reaches a production virtual machine or production environment.

If you have a container and a container runtime, your app can be deployed nearly anywhere. While there are some caveats to this statement, such as the need to build separate images for x86 and ARM platforms, containerized applications can move across hosting locations almost as easily as the hosting companies can charge your wallet. Need an on-premise production system? Great, you can use containers. Need to move those containers to achieve more scale by using the cloud and without spending anything on new hardware? Great, you can use the *same* containers. Need to distribute the application to other parts of the world with a load balancer? Great, you can use the *same* containers. Need to move the application to a different cloud provider? Great, you can use the *same* containers. The list goes on while the point is made: containers are the most portable form of application deployment.

While containers solve a massive challenge in deploying applications, the technology was not the first to do so. Popularized by Heroku and other platform-as-a-service (PaaS) businesses, buildpacks enable application developers to push their code via Git, and then the buildpack tool automatically configures the server or virtual machine. With buildpacks, developers would create a simple configuration file, called a *Procfile*, that contains a single line to run the app itself (e.g., `web: gunicorn main:app`). Beyond the *Procfile*, buildpacks use the code itself to *infer* how to configure the server. This often means that if `requirements.txt` is found in the code, Python would be used. If it is `package.json`, Node.js would be used. Notice that I purposefully omitted the version of the application runtime because not all buildpacks support all versions of application runtimes. This is why we need a better solution to help with portability and deploying our applications.

While buildpacks are powerful, the OS-level runtime configuration is largely *inferred* instead of *declared* as we do with Dockerfiles or infrastructure as code (IaC) tools like Ansible, SaltStack, or Puppet. What's more, buildpacks do not always support modern runtimes (e.g., Python 3.11 or Node.js 18.17 LTS, etc.), which, as we now know, containers do very well. That said, buildpacks paved the way for containers to become the most portable form of application deployment. Luckily for us, we can also use buildpacks to create container images using an open-source tool called pack (<https://buildpacks.io/>). Using buildpacks is outside the scope of this book, but it's important to note that the challenge of portability has been around for a long time.

Buildpacks and containers ushered in a new era of software development that enabled the field of DevOps to flourish. With these tools and cloud-based hosting services, developers have been able to create and deploy more production-ready applications than ever before, often as solo developers or very small teams. Adding the fact

that more businesses have adopted all kinds of software (cloud-based or otherwise), we continue to see a rising need for managing and deploying more public-facing and private-facing applications. This is where Kubernetes comes in.

Once we have built container images and access to a Kubernetes cluster, deploying an application is as simple as writing a YAML configuration file known as a Kubernetes manifest.

8.1 **Getting started with Kubernetes**

Kubernetes is designed to work across a cluster of computers, whether it's cloud-based virtual machines, a rack of servers you have on-premises, a group of Raspberry Pis, or even repurposed desktop computers that were sitting dormant in the back of your closet somewhere. All of these options can run a version of Kubernetes that then runs all of your various containers. So which to pick? Self-service or managed Kubernetes?

8.1.1 **Self-service or managed Kubernetes?**

While Kubernetes is open-source and we could manage it ourselves, I prefer using cloud-based managed Kubernetes for the following reasons:

- *Public static IP addresses*—Managed Kubernetes can provision new public static IP addresses in seconds (beyond your virtual machines). While you can do this with self-serviced K8s, it's not as easy, fast, or simple.
- *Volumes and storage*—In fewer than 10 lines of YAML, we can reserve a volume for a managed K8s cluster. We can reserve as many volumes as our cloud provider allows our account. Once again, we can do this in self-serviced K8s; it's just not as easy, fast, or simple.
- *Cloud-based*—Hardware hosts, power costs, cooling, configuration, and all the things that go with the cloud are nonexistent using cloud-based managed Kubernetes. Costs are predictable, the hardware is reliable, outages are short-lived and not our responsibility, and so on.
- *Simplicity*—Using Akamai Connected Cloud to provision a Kubernetes cluster is nearly as simple as provisioning multiple virtual machines.
- *Speed*—Akamai Connected Cloud can create your K8s clusters in just a few minutes.
- *Control plane*—The K8s control plane is responsible for managing the cluster and the virtual machines that run the containers. In other words, it's the brains of the operation. When you use a cloud-based managed solution, the Kubernetes Control Plane is managed by the service provider at little to no additional cost.
- *IaC enabled*—Using an Infrastructure as Code (IaC) tool like Terraform can turn on or turn off a managed Kubernetes cluster in a few minutes. While IaC tools can also help configure a self-managed cluster, it's not as easy.

With this in mind, it's important to understand that Kubernetes is designed to be as portable as possible, which makes migrating from one Kubernetes cluster to another relatively easy. The parts that are tricky are managed services like volumes, load balancers, and related public IP addresses. These services are typically cloud-specific and require additional configuration to migrate successfully. The portability all lies in what

we will be spending a lot of this chapter on: Kubernetes manifests. With this in mind, let's provision a Kubernetes cluster using Akamai Connected Cloud.

8.1.2 Provisioning a Kubernetes cluster

Here are the steps to provision a Kubernetes cluster using Akamai Connected Cloud:

- 1 Log in to Akamai Connected Cloud.
- 2 Click the Kubernetes tab.
- 3 Click the Create Cluster button.
- 4 Create an account on Akamai Connected Cloud at linode.com/rk8s (book readers receive a promotional credit as well).
- 5 Log in to Akamai Connected Cloud (formally Linode).
- 6 Click the drop-down Create > and select Kubernetes.
- 7 Use the following settings:
 - Cluster label—rk8s-cluster
 - Region—Seattle, WA (or the region nearest you)
 - Version—1.26 or whatever is the latest
 - HA Control Plane (High Availability)—No (or Yes if you want to pay about \$60 more per month)
- 8 Also do the following:
 - Add Node Pools
 - Select Shared CPU
 - Select Linode 2 GB (or larger)
 - Ensure the count is 3 or greater (default is 3).

Before you click *Create Cluster*, verify your screen resembles figure 8.1.

Kubernetes / Create Cluster

Cluster Label: rk8s-cluster

Region: Seattle, WA (us-sea)

Kubernetes Version: 1.26

HA Control Plane: No

Add Node Pools: Shared CPU

Plan	Monthly	Hourly	RAM	CPUs	Storage
Linode 2 GB	\$12	\$0.018	2 GB	1	50 GB

Cluster Summary: Linode 2 GB Plan, 1 CPU, 50 GB Storage, \$36.00/month, Create Cluster

Figure 8.1 Preprovisioned Kubernetes cluster options

The Linode Kubernetes Engine (LKE) will automatically provision the virtual machine instances based on our previous configuration. These particular instances do *not* need our SSH keys because we will not be logging directly into them. The virtual machine instances will appear in your Linode dashboard just like any other instances. LKE will manage all related virtual machines and the Kubernetes Control Plane for us.

Click *Create Cluster*, and after a few moments, we'll see the LKE detail view for this cluster (located at <http://cloud.linode.com/kubernetes>). The result you will see is in figure 8.2.

The screenshot shows the Linode Kubernetes Engine (LKE) dashboard for a cluster named 'rk8s-cluster'. The top section displays cluster details: Version 1.26, 3 CPU Cores, 6 GB RAM, \$36.00/month, 150 GB Storage, and the Kubernetes API Endpoint: <https://8cd52046-a16f-491b-8c71-38d20c37e040.us-east-2.linode.lke.net:443>. It also shows the Kubeconfig file 'rk8s-cluster-kubeconfig.yaml' with options to View or Reset. The bottom section, titled 'Node Pools', shows a single pool named 'Linode 2 GB' with three nodes in a 'Provisioning' state. Each node has a unique Linode ID and a public IP address.

Linode	Status	IP Address	Recycle
lke127692-189305-64efac166e59	Provisioning	172.232.160.218	Recycle
lke127692-189305-64efac169fcb	Provisioning	172.232.160.254	Recycle
lke127692-189305-64efac16ceb2	Provisioning	172.232.160.239	Recycle

Figure 8.2 Linode Kubernetes Engine provisioning cluster

A few key aspects to note:

- The cluster is named `rk8s-cluster` (or whatever you named it).
- A kubeconfig file has been created for you (`rk8s-cluster-kubeconfig.yaml`). This file contains the permissions necessary to access your root service account in your Kubernetes cluster. You can download, view, or reset the values of this file. Resetting the kubeconfig file is much like resetting a password.
- Node pools are an advanced feature in Kubernetes that allows us to configure additional nodes (virtual machines) in our cluster after the initial provisioning. Modifying node pools is outside the scope of this book, but it's important to note that we can add additional nodes to our cluster at any time.
- As expected, we see public IP addresses on each virtual machine (node) in the node pool.
- Kubernetes API endpoints and the GUI are accessible through this URL (more on this soon).

8.1.3 Core concepts and components

Before we dive into using Kubernetes, there's a number of core concepts I would like for you to understand at a high level. These concepts are not exhaustive for Kubernetes but are going to be the focus of this chapter. These concepts will stick better as you start using Kubernetes and provisioning the various resources on your cluster. Figure 8.3 shows a high-level visualization of these concepts.

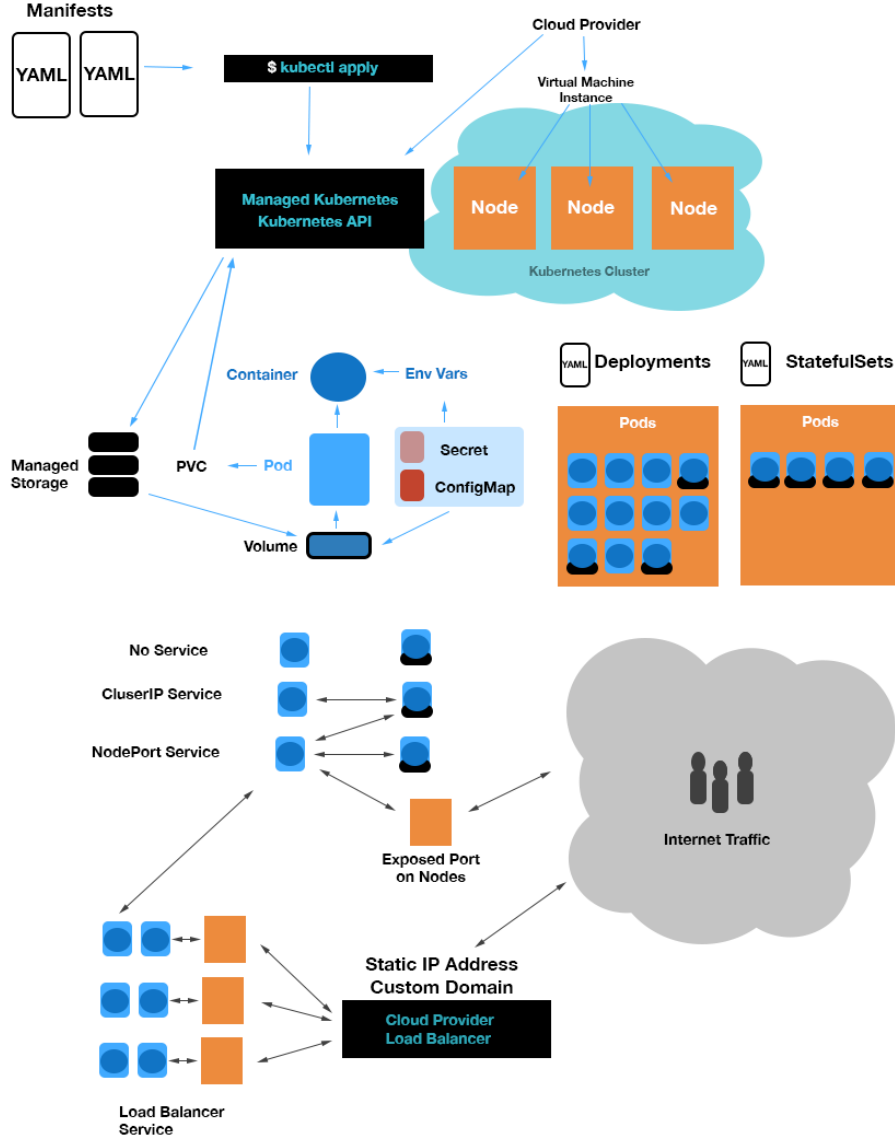


Figure 8.3 Kubernetes concepts

Here are the terms:

- *Cluster*—A group of virtual machines that run Kubernetes and our containers. While clusters can be run on other kinds of compute devices, we will focus on virtual machines in this book.
- *Node*—A virtual machine that is a member of our Kubernetes cluster.
- *Container*—A portable application bundle that runs in isolation via a container runtime. Kubernetes uses a lightweight container runtime called containerd. As a reminder, Docker Engine is a container runtime that Docker Compose uses.
- *Pod*—This is the smallest unit of running a container, including the configuration the container needs. Pods can also define multiple containers that share resources (e.g., CPU, memory, etc.). While we can create Pods directly, it's more common to use a Deployment to create Pods.
- *Deployment*—Deployments define the conditions to run containers via Pods. In a Deployment, we can specify the container image, the number of Pod replicas we want to run, and more. Deployments are the most common way to run containers in Kubernetes.
- *Service*—Services expose our Pods (and thus our containers) to either our Kubernetes network (internal) or to the internet (external). We'll use Services in conjunction with Deployments to expose our containers to the internet with cloud-based load balancing.
- *Namespaces*—Namespaces are a way to organize our Kubernetes resources. A Namespace can be assigned to any Kubernetes resource (e.g., a Deployment, a Service, a Pod, etc.), allowing us to filter our resources by Namespace. You can use namespaces for helping to keep projects separate, for app environment staging (e.g., production, preproduction, dev, rnd, etc.), for different organizations, or for internal teams.
- *Secrets* and *ConfigMaps*—These are the mechanisms built into Kubernetes for storing key-value pairs (much like the dotenv files we know and love). Secrets are really just base64-encoded values, while ConfigMaps are key-value pairs. Both are used to inject environment variables into containers. I tend to think of these together because Kubernetes Secrets, while convenient, are not very secure. We'll cover both in this chapter.
- *StatefulSets*—Deployments, like containers, are designed to be ephemeral. StatefulSets are designed to be more permanent and are often used for databases (e.g., PostgreSQL, Redis, MySQL, etc.) and other stateful applications.
- *Volumes*—When we need data to persist beyond a container's lifecycle, we can attach persistent volumes managed directly from our cloud provider.
- *Service account*—Kubernetes does not have *users* in the traditional sense; instead, it has *service accounts*. Service accounts are used to authenticate with the Kubernetes API and manage or access various Kubernetes resources. When you provision

a new managed cluster, you typically get a root service account with unrestricted permissions to your Kubernetes cluster. This root service account also comes with sensitive access keys through an automatically generated kubeconfig file. While creating service accounts is outside the scope of this book, it can be important as your team or Kubernetes project grows in contributors.

- *Kubeconfig*—With managed clusters, the root service account is provisioned with a kubeconfig file (e.g., `rk8s-cluster-kubeconfig.yaml`) that contains the sensitive access keys to your Kubernetes cluster. Treat this file like any other secret (e.g., only share with trusted individuals, reset the keys if you think it's been compromised, do not check it in to Git, etc.). The kubeconfig file is used to authenticate with the Kubernetes API to access a specific Service Account. We will create a new kubeconfig file in this chapter.
- *kubectl*—The Kubernetes Command Line interface is called `kubectl` and is variously pronounced “cube CTL,” “cube control,” and “cube cuddle.” In conjunction with a service account via a kubeconfig file, we’ll use `kubectl` to manage our Kubernetes cluster, deploy containers, manage Services, create Secrets, restart Deployments, and more. `kubectl` is a powerful tool, and we’ll use it a lot in this chapter.

With this in mind, let’s learn a few ways to connect and interact with our new Kubernetes Cluster.

8.2 Connecting to Kubernetes

Kubernetes has two primary ways to connect to a cluster: the Kubernetes Dashboard GUI and the Kubernetes CLI (`kubectl`). Let’s start with the GUI.

8.2.1 The Kubernetes Dashboard GUI

The Kubernetes Dashboard is an easy way to review what is happening in your Kubernetes Cluster as well as manage various resources like Pods, Deployments, Services, and more.

Personally, I tend to use `kubectl` far more than the K8s Dashboard, but it’s still a useful tool for rapid experimentation. To access the Dashboard, do the following:

- 1 Log in to Akamai Connected Cloud.
- 2 Navigate to Kubernetes.
- 3 Select your cluster (`rk8s-cluster`).
- 4 Download the kubeconfig file (`rk8s-cluster-kubeconfig.yaml`).
- 5 Click the Kubernetes Dashboard button.

These steps will open a new tab in your browser and prompt you to log in with your kubeconfig file. To do so, select Kubeconfig (not token), choose your `rk8s-cluster-kubeconfig.yaml` download file, and click Sign In, and the result should resemble figure 8.4.

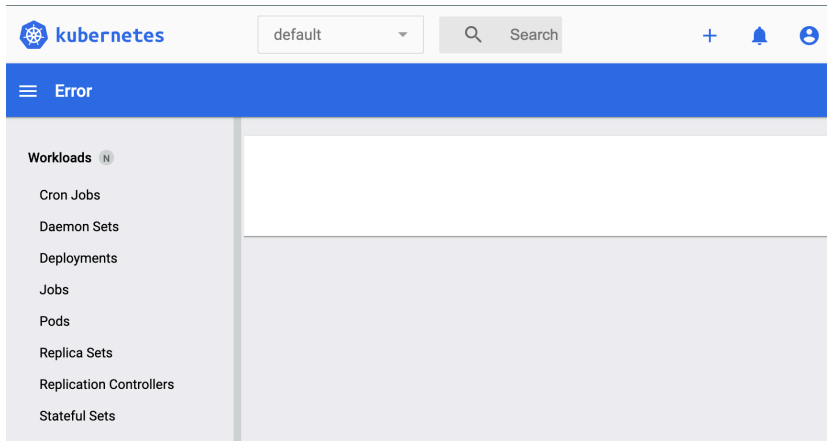


Figure 8.4 Kubernetes Dashboard

This Dashboard is painfully empty and has no wizard-like forms to help us configure our cluster or K8s resources. This is by design. To configure Kubernetes we must use specific YAML files known as Kubernetes manifests. If you hit the plus sign (+) in the top right corner, you can paste in manifests to create resources. We will learn more about manifests soon. Even with no resources provisioned, we can view the current nodes available in your cluster as you can review in the *Nodes* tab, which will look much like figure 8.5.

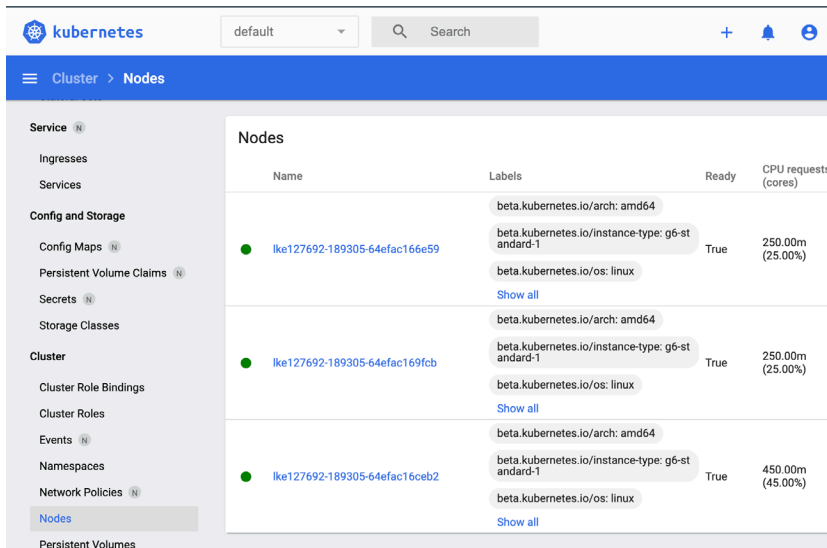


Figure 8.5 Kubernetes Dashboard Nodes

While some users might prefer using the dashboard to better visualize what is happening in their cluster, I prefer using `kubectl` because it's a tool I can use in my terminal and automate with custom scripts and in my CI/CD pipelines. Now that we have reviewed the dashboard, let's install `kubectl` on our local machine.

8.2.2 Installing `kubectl`

Before we install `kubectl`, it's important to note that we will not have a local version of Kubernetes installed on our machine. `kubectl` automatically attempts to connect to a *localhost* version of Kubernetes, as you will see shortly. Unlike Docker Desktop, `kubectl` is primarily installed on the command line (e.g., Terminal or PowerShell) and not as a GUI application. Like many tools, there are a number of ways to install `kubectl`, depending on your operating system. Consider reviewing the official `kubectl` install documentation (<https://kubernetes.io/docs/tasks/tools/>) if you need to understand the other ways to install `kubectl` on various operating systems.

Once `kubectl` is installed, you can verify it with the command `kubectl version --client --output=yaml`. The result should resemble figure 8.6.

```
% kubectl version --output yaml
clientVersion:
  buildDate: "2023-01-18T15:51:24Z"
  compiler: gc
  gitCommit: 8f94681cd294aa8cfd3407b8191f6c70214973a4
  gitTreeState: clean
  gitVersion: v1.26.1
  goVersion: go1.19.5
  major: "1"
  minor: "26"
  platform: darwin/arm64
kustomizeVersion: v4.5.7

The connection to the server localhost:8080 was refused
```

Figure 8.6 `kubectl version`

Figure 8.6 shows that I have `kubectl` installed and the connection to server `localhost:8080` failed. This error is expected as we do not have `kubectl` configured, and there is likely not a local version of Kubernetes running (for sure, not at port 8080). I show you this output up front so you know what to expect when you install `kubectl` on your machine.

To install `kubectl`, there are three primary options we'll use based on your operating system:

- On macOS and using package manager Homebrew.
- On Windows and using the package manager Chocolatey.
- On Linux, like on a virtual machine or the windows subsystem for linux (WSL), we will download the binary version of `kubectl` and install it manually.

The methods I provide in the following sections are the ones I prefer because they tend to be the easiest to get up and running.

INSTALL KUBECTL WITH HOMEBREW ON MACOS

Homebrew is an open-source package manager for macOS and some Linux distributions. Homebrew makes installing a lot of third-party software much easier, including `kubectl`. Installing `kubectl` with Homebrew is as simple as the following:

- 1 Run `brew update` in Terminal to update Homebrew. If the `brew` command fails, install homebrew from <https://brew.sh/>.
- 2 Run `brew install kubectl`.
- 3 Verify `kubectl` with `kubectl version --client --output=yaml`.

If you are having problems with Homebrew, consider using the `kubectl` binary installation as seen in the section *Install the Kubectl Binary on Linux*. If you're using Apple Silicon, be sure to use the ARM version (e.g., `export PLATFORM="arm64"`). If you're not on Apple Silicon, use the x86 version (e.g., `export PLATFORM="amd64"`).

INSTALL KUBECTL WITH CHOCOLATEY ON WINDOWS

Chocolatey is an open source package manager for Windows that makes installing a lot of third-party software easy. We can use Chocolatey to install `kubectl` and many other tools. Here's how to install `kubectl` with Chocolatey:

- 1 Run `choco upgrade chocolatey` in PowerShell to update Chocolatey. If the `choco` command fails, install Chocolatey from <https://chocolatey.org/install>.
- 2 Run `choco install kubernetes-cli`.
- 3 Verify `kubectl` with `kubectl version --client --output=yaml`.

If you are having problems with Chocolatey, consider using the Windows subsystem for Linux (WSL) and follow the instructions in the next section.

INSTALL THE KUBECTL BINARY ON LINUX

Installing the `kubectl` binary on Linux requires a few more steps than using package managers. The steps in listing 8.1 come directly from the Official `kubectl` Documentation (<https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/>) but are modified to easily switch platforms (e.g., `amd64` or `arm64`) for different system architectures.

Listing 8.1 Installing `kubectl` with `apt` on Linux

```
# if x86, use amd64
export PLATFORM="amd64"

# if ARM, use arm64 and
```

```
# uncomment the following line
# export PLATFORM="arm64"

# get latest release version
export VER=$(curl -L -s https://dl.k8s.io/release/stable.txt)

# download latest release
curl -LO "https://dl.k8s.io/release/$VER/bin/linux/$PLATFORM/kubectl"

# verify the release
curl -LO "https://dl.k8s.io/$VER/bin/linux/$PLATFORM/kubectl.sha256"
echo "$(cat kubectl.sha256) kubectl" | sha256sum --check

# install kubectl
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

# verify installation
kubectl version --client --output=yaml
```

8.2.3 Configuring kubectl

Now that we have a Kubernetes cluster provisioned, a kubeconfig file downloaded, and kubectl installed, we can configure kubectl to connect to our cluster. To do this, we will use the kubeconfig file we downloaded from Akamai Connected Cloud.

To configure kubectl, we can either

- Use `~/.kube/config` as the path for our kubeconfig file. This means we would name the file `config` (without `.yaml`) and store it in the `.kube` directory for our user.
- Use the environment variable `KUBECONFIG` with the absolute path to our kubeconfig file. This means we can name the file anything we want and store it anywhere we want.

We will use the `KUBECONFIG` variable for local development to make it easier to use multiple kubeconfig files across multiple Kubernetes projects.

Using `KUBECONFIG` is as simple as the following:

- Terminal (*macOS/Linux/WSL*)—`KUBECONFIG=/path/to/kubeconfig.yaml kubectl get nodes`
- PowerShell (*Windows*)—`setx KUBECONFIG "C:\path\to\kubeconfig.yaml" && kubectl get nodes`

Writing the entire environment variable each time can be tedious, so we can set the environment variable for each command-line session with

- Terminal (*macOS/Linux/WSL*) —`export KUBECONFIG=/path/to/kubeconfig.yaml`
- PowerShell (*Windows*)—`setx KUBECONFIG "C:\path\to\kubeconfig.yaml"`

and then simply `kubectl get nodes`.

Once again, we'll have to remember to set the `KUBECONFIG` environment variable each time we open a new Terminal or PowerShell session. While we could set the `KUBECONFIG` environment variable globally, it defeats the purpose of using multiple kubeconfig files for different projects.

If you're anything like me, you'll want to set up your local development environment to use multiple clusters right away. While this does take more configuration, it really sets you up for success as you learn Kubernetes and adapt it for your projects.

One of the reasons I use Visual Studio Code (VSCode) as my text editor is how simple it is to configure the VSCode terminal to use pre-defined environment variables. As a reminder, text editor terminals (including PowerShell-based ones) are not the same as the operating system's Terminal or PowerShell. While other text editors may have similar features, I'll show you how to configure VSCode to use multiple kubeconfig files for different projects.

Within VSCode, you can update your `.code-workspace` file to include the *terminal.integrated.env* setting for each platform with specific *key/value* pairs. For example, you can use `terminal.integrated.env.osx` to set `PORT` to 8080 or `KUBECONFIG` to `some/path/rk8s-cluster-kubeconfig.yaml`. With these values set and your VSCode workspace open, you can open a new VSCode terminal session, and these values will be automatically injected into your command line, which you can verify by running `echo $YOUR_VARIABLE` like `echo $KUBECONFIG`. Listing 8.2 shows us cross-platform examples in a `.code-workspace` file that sets multiple environment variables (i.e., `KUBECONFIG`, `PY_PORT`, `JS_PORT`) for each platform.

Listing 8.2 Setting KUBECONFIG in VSCode

```
{
  ...
  "settings": {
    ...
    "terminal.integrated.env.osx": {
      "KUBECONFIG": "${workspaceFolder}/.kube/kubeconfig.yaml",
      "PY_PORT": "8080",
      "JS_PORT": "3000"
    },
    "terminal.integrated.env.windows": {
      "KUBECONFIG": "${workspaceFolder}\\\\.kube\\kubeconfig.yaml",
      "PY_PORT": "8080",
      "JS_PORT": "3000"
    },
    "terminal.integrated.env.linux": {
      "KUBECONFIG": "${workspaceFolder}/.kube/kubeconfig.yaml",
      "PY_PORT": "8080",
      "JS_PORT": "3000"
    },
  },
}
```

The variable `${workspaceFolder}` is special for VSCode and will be replaced by the absolute path to the workspace. Using the line `${workspaceFolder}/.kube/kubeconfig.yaml` results in the path *relative* to the workspace's root (often the same folder that holds the `.code-workspace` file), which is also true for the Windows version (*terminal.integrated.env.windows*). If you prefer absolute paths, you can do the following:

- On macOS or Linux—`${workspaceFolder}/.kube/kubeconfig.yaml` to `/path/to/my/kube/config.yaml`
- On Windows—Swap `"${workspaceFolder}\\.\kube\\kubeconfig.yaml"` to `"C:\path\to\my\kube\config.yaml"`. Notice the double backslashes.

Regardless of how you set the `KUBECONFIG` environment variable, you should be able to verify `kubectl` can connect with `kubectl get nodes` and see the nodes in your cluster. The result should resemble figure 8.7.

```

% kubectl get nodes
NAME                                STATUS    ROLES    AGE   VERSION
lke127692-189305-64efac166e59      Ready    <none>    21h   v1.26.3
lke127692-189305-64efac169fcb      Ready    <none>    21h   v1.26.3
lke127692-189305-64efac16ceb2      Ready    <none>    21h   v1.26.3

```

Figure 8.7 `kubectl` Get Nodes

Now that we have `kubectl` configured, let's deploy our first container.

8.3 Deploy containers to Kubernetes

With Kubernetes and `kubectl` configured, we can now start the process of creating resources within Kubernetes. Our goal with deploying containers is to have a number of manifest files that describe the resources we are trying to configure. These resources range from the container itself to a number of other features that support how the container needs to run.

Before we start working with manifests, let's create a new folder for all of our Kubernetes manifests. In my case, I will create the folder `~/dev/roadtok8s-kube` that will contain the following files:

- `.kube/kubeconfig.yaml`—The `kubeconfig` file we downloaded from Akamai Connected Cloud. The path for this will equate to `~/dev/roadtok8s-kube/.kube/kubeconfig.yaml`.
- `roadtok8s-kube.code-workspace`—Run for my VSCode workspace file that contains my `KUBECONFIG` environment variable for the path to my `kubeconfig` file.
- `my-first-pod.yaml`—As of now, this will be an empty file, but we will add to it shortly.

With our `roadtok8s-kube` directory ready, let's create your first Pod manifest to run your first container image.

8.3.1 Your first Pod and manifest

Kubernetes has many different resource types. Each resource type is declared within your manifest using the format `kind: <resource>` or, in our case now, `kind: Pod`. In this section, we will use the Pod resource to run a single instance of the NGINX container image. Pods are the smallest configurable unit in Kubernetes, so they are a great candidate for our first manifest. Here are a few key features that all Kubernetes manifests must include:

- `apiVersion`—The version of the Kubernetes API to use. This is typically set to `v1` for most resources.
- `kind`—The type of resource to create. There are a lot of resource options, but we'll stick with `Pod` and `Service` for this section and expand to others later in this chapter.
- `metadata.name`—The metadata for any resource helps us describe that resource (e.g., names, labels, etc.). The `name` field within the `metadata` block is required for all resources.
- `spec`—This is the block we define our specification(s) for this particular resource. This chapter will have a number of specification examples for different resources. This block is the most important part of any manifest and has by far the most potential options.

With this in mind, let's create your first manifest file running a single NGINX-based Pod. If you haven't done so already, create the file `my-first-pod.yaml` and add the contents from the following listing, which has a minimal manifest for a Pod that runs the Docker-managed public NGINX container image.

Listing 8.3 K8s YAML manifest for an NGINX Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: first-pod-nginx
spec:
  containers:
  - name: nginx
    image: nginx
```

Now that we have this file, we can do a few things with it:

- `kubectl apply -f my-first-pod.yaml`—This command will create the Pod in our Kubernetes cluster. If the Pod already exists, it will update the Pod with any changes in the manifest.
- `kubectl delete -f my-first-pod.yaml`—This command will delete the Pod from our Kubernetes cluster if it exists.

If you are using the Kubernetes Dashboard, you could use this file to create a new Pod directly in the dashboard. Doing so is outside the scope of this book, but I encourage you to explore if you are interested.

After running the `kubectl apply` command, we can verify the resource was created by running `kubectl get pods`, which should result in figure 8.8.

```
% kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
first-pod-nginx	1/1	Running	0	4s

Figure 8.8 `kubectl get pods`

The status of any Pod, known as a Pod’s *phase*, can be one of the following:

- *Pending*—The Pod is being created and is not yet running. This often means the container is being pulled from a container registry *or* is still booting.
- *Running*—The Pod is running, and thus the Pod’s container(s) is running as well.
- *Succeeded*—For Pods that do not run continuously, this status indicates the Pod has completed successfully.
- *Failed*—Not surprisingly, this status indicates the Pod has failed.
- *Unknown*—For some reason, the state of the Pod could not be obtained.

You should expect that the NGINX container image will be running successfully after a couple of minutes. If it’s not running, you can investigate the Pod by using `kubectl describe pod <pod-name>` or `kubectl logs <pod-name>`. The NGINX container image is very mature, and thus, errors are unlikely to happen because of the container itself (this may or may not be true of the containers we built for our Python and Node.js projects).

Now that the container is running because of the Pod resource, how do we actually visit the internet-facing NGINX default page from this Pod? Enter your first Service resource.

8.3.2 Your first Service

Much like unpublished containers on virtual machines, Pods have no way to reach the outside world without some additional configuration. Pod-to-Pod communication is also impossible without additional configuration.

Before we get to internal cluster communication, let’s do something more exciting: exposing our Pod to the internet via the Kubernetes Service resource. Seeing the Welcome to nginx! page is a beautiful thing on a newly minted Kubernetes Cluster. To do this, you need to create a new manifest with the Service resource (`kind: Service`) to expose our NGINX Pod to the internet.

Before we create the Service resource, let’s review a few of the fundamental types of Kubernetes Services:

- *NodePort* (external traffic)—Our cluster has three nodes (virtual machines), and each node has a static public IP address provided by the ISP and Akamai Connected Cloud. Using the NodePort Service resource allows you to *expose* a Pod to a specific port value, such as 30007, on every available node in your K8s cluster.
- *LoadBalancer* (external traffic)—This is one of my favorite Service types because it automatically provisions a cloud-based load balancer with a static public IP address for a K8s Service resource. In the case of Akamai Connected Cloud, the Linode Kubernetes Engine (i.e., the managed Kubernetes Service we’re using) will provide us with a public static IP address through the cloud-based load balancing service called Linode NodeBalancers.
- *ClusterIP* (internal cluster-only traffic)—Using this Service type allows other containers in our Kubernetes Cluster to communicate with our Pods (or Deployments), enabling container-to-container communication within our cluster. More accurately, ClusterIPs are for Pod-to-Pod or Deployment-to-Deployment communication within our cluster. ClusterIPs are great for Pods or Deployments that do not need to be exposed to the internet, such as databases, internal APIs, and others.

We will start with *NodePort* for our Pod because it’s the easiest to test quickly without adding anything else to our cluster or our cloud provider.

Before you create a service resource, you need to update your pod manifest to make it easier for a Service to find and attach to it. To do this, add a label to the manifest of your Pod (your-first-pod.yaml) within the metadata block, just like listing 8.4.

Listing 8.4 K8s YAML manifest for an NGINX Pod with a Label

```
apiVersion: v1
kind: Pod
metadata:
  name: first-pod-nginx
  label:
    app: my-nginx
...
```

While we typically add metadata labels when we first create resources like Pods, I wanted to add additional emphasis to the importance now: if your Pod or Deployment does not have a label, the service resource will not work correctly.

With this in mind, update your pod by running `kubectl apply` on the updated manifest with `kubectl apply -f my-first-pod.yaml`. Now run `kubectl get pods`, and your pod should still be running. Run `kubectl get pods my-nginx -o yaml` to add even more details about your pod (including verifying that your label is present in the output).

Within any Kubernetes Manifest, regardless of its type (Pod, Service, Deployment, etc.), we can apply a metadata label to the resource. Metadata labels are key-value pairs that can be used to filter any Kubernetes Resources. In the case of your updated Pod

resource and the `app: my-nginx` label, you can see the filtering in action with the command `kubectl get pods -l app=my-nginx`. Running this command will either result in a list of Pods matching this selector or no Pods at all.

The selector, or filter, for narrowing the list of results of any given Kubernetes Resource is the `-l` or `--selector` flag as in `kubectl get <resource> -l <key>=<value>` or `kubectl get <resource> --selector <key>=<value>`. Here are a few examples of the selector filter in action:

- `kubectl get pods -l my-label=my-value`
- `kubectl get pods -l app=my-nginx`
- `kubectl get pods --l app=nginx`
- `kubectl get services -l app=my-nginx`
- `kubectl get deployments -l sugar=spice`
- `kubectl get deployments -l maverick=pilot`

As these examples imply, the metadata label can be anything you want; that goes for both the key and its related value. In other words a resource's metadata label key value (e.g., `app:` in our Pod manifest) has no special meaning to Kubernetes; it's just a conventional name. While some metadata *can* have special meaning, this one does not. Just remember the key-value pair you use (e.g., `buddy: holly`, `hello: world`, `peanut: butter`), will be known as the resource's *selector* value.

Now that we have seen selectors in action based on metadata key-value pairs, let's define a few key declarations we need for our Services within the `spec` block:

- `type: <service-type>`—Select the Kubernetes Service type you want to use: `NodePort`, `LoadBalancer`, or `ClusterIP`.
- `selector: <labeled-metadata-key-value-pair>`—The selector is the most important part of our Service. The Service will automatically load balance requests based on the *matching* key-value pairs defined here. If the key-value pair used yields no filtered results (e.g., `kubectl get pods -l incorrect:data`), the Service is useless.
- `ports`—If you recall back to our Docker publish argument (`-p` or `--publish`), we specified the port we wanted to expose and the port we wanted to map to (e.g., `-p <expose>:<mapping>`). In the case of `kind=NodePort`, you define three port values: `port`, `targetPort`, and `nodePort`. *port* is so that internal cluster-based communication can access this Service. *targetPort* is the port of the container running as defined by the Pod or Deployment (yes, just like the mapped port in Docker). *nodePort* is the port we want to expose on every node in our cluster (e.g., `http://<node-static-public-ip>:<nodePort>;`).

Now let's do a quick check before we define our Service:

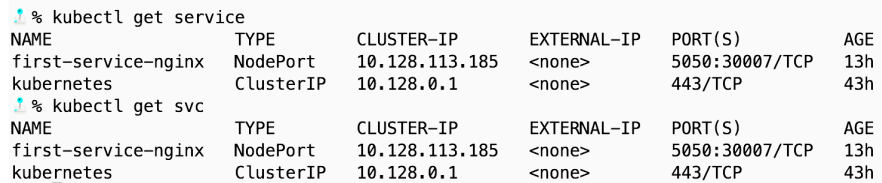
- The port we'll use on our nodes is 30007 (e.g., `nodePort: 30007`).
- The port we'll use for internal cluster communication is 5050 (e.g., `port: 5050`). We'll use internal cluster communication later in this chapter.
- The port the container expects is the port we'll use for our `targetPort` (e.g., `targetPort: 80`).
- Our Pod has a label with `app: my-nginx`, so our Service can use the selector `app: my-nginx` to find our Pod.

With this in mind, create a file named `my-first-service.yaml` in your `roadtok8s-kube` directory with the contents of the following listing.

Listing 8.5 K8s YAML manifest for an NGINX Service

```
apiVersion: v1
kind: Service
metadata:
  name: first-service-nginx
spec:
  type: NodePort
  ports:
    - port: 5050
      targetPort: 80
      nodePort: 30007
  selector:
    app: my-nginx
```

Provision this Service with `kubectl apply -f my-first-service.yaml`. Once you do, you can verify the Service is available and listed with `kubectl get service` or the shortcut version `kubectl get svc`. The result should resemble figure 8.9.



```
% kubectl get service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
first-service-nginx	NodePort	10.128.113.185	<none>	5050:30007/TCP	13h
kubernetes	ClusterIP	10.128.0.1	<none>	443/TCP	43h

```
% kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
first-service-nginx	NodePort	10.128.113.185	<none>	5050:30007/TCP	13h
kubernetes	ClusterIP	10.128.0.1	<none>	443/TCP	43h

Figure 8.9 `kubectl get service`

The listed Services should match our `first-service-nginx` and the default Kubernetes Service. The default Kubernetes Service is how our cluster communicates with the managed Kubernetes API; this API, known as the control plane, is how resources work. The control plane is what schedules and runs resources across the cluster. With managed Kubernetes, the control plane is managed by the service provider but is still essentially the glue that keeps it all together. The `kubectl` tool communicates directly to the Kubernetes API with this Service. For our Service, here's what's important to note:

- **Type**—Verify you see `NodePort` for our Service.
- **Cluster IP**—This is the IP address value for internal Pod-to-Pod (or Deployment-to-Deployment) cluster communication.
- **External IP**—None are listed because we use `NodePort` and not `LoadBalancer`. If it was `LoadBalancer`, a public static IP address would be provisioned for us.
- **Ports**: `5050:30007`. This means that internal cluster communication can access this Service on port 5050, and external traffic can access this Service on port 30007 for each node.

Now, let's verify that `NodePort` actually works by getting each public static IP address for each node. To do this, you can

- Log in to the Akamai Connected Cloud Console and review it.
- Run `kubectl get nodes -o wide` command (which will list internal and external IP addresses).

Once you find a node's public static IP address, visit `http://<your-node-public-ip>:30007` and see the NGINX hello world in all its glory. Congratulations, you have just deployed your first internet-connected container on Kubernetes.

Job done, right? Not yet. Right now you have one container running because of one Pod; we need to do what Kubernetes was built for: scale our containers up.

8.3.3 From Pods to Deployments

The Deployment resource is the most common way to deploy containers to Kubernetes because it enables us to scale our Pods and thus our containers. Deployments allow us to have multiple instances of the same container application running across our cluster (yes, across each node and virtual machine). Once we scale our Deployments, our Service resource will automatically load balance traffic to each Pod that is available and ready for traffic.

Deployments are basically groups of Pods, and since they are groups of Pods, a Deployment manifest is basically a Pod manifest with a few additional declarations in the `spec` block:

- **replicas**:—This is the number of Pods we want to run with this Deployment. If unspecified, it defaults to 1.
- **selector**:—With Deployments, we predefine the label selector to match our pods. This selector will also be used in the Pod template (see following discussion).
- **template**:—This is exactly the Pod manifest without the `kind` or `apiVersion` declarations. You can copy and paste the `metadata` and `spec` values. Within the `metadata` values, ensure the label's key-value pair matches the selector in the Deployment's `selector` block (not the Service yet).
- **apiVersion**: While this is not in the `spec` block, we use `apps/v1` as the `apiVersion` for Deployments. This is the only `apiVersion` we will use for Deployments in this book.

With this in mind, let's create a new file called `my-first-deployment.yaml` in your `roadtok8s-kube` directory with the contents of the following listing.

Listing 8.6 K8s YAML manifest for an NGINX Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 15
  selector:
    matchLabels:
      app: my-nginx
  template:
    metadata:
      labels:
        app: my-nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
```

With this manifest, apply the changes with `kubectl apply -f my-first-deployment.yaml`. Once you do, you can verify the Deployment with `kubectl get deployments` or the shortcut version `kubectl get deploy`. The result should resemble figure 8.10.

```
↑ % kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
rk8s-deployment    15/15   15           15          14s
↑ % kubectl get deploy
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
rk8s-deployment    15/15   15           15          16s
```

Figure 8.10 `kubectl get deployment`

In this example, I put 15 replicas to illustrate how simple it is to scale up the number of Pods running because of the Deployment resource and once again highlight how the selector filter works. If you run `kubectl get pods --selector app=my-nginx`, you should see at least 15 Pods running (and possibly 16 with our previous Pod resource manifest).

The replica count is the *desired state* for Kubernetes Deployment resources. This means that Kubernetes will continuously deploy Pods until 15 of them are running and healthy, as defined in our manifest. If any Pods fail at any time, Kubernetes will automatically roll out new Pods until the number of Pods reaches the replica count. The rollout process is tiered by default, so if you were to update a Deployment manifest, only a few Pods would be updated at a time. This is a great feature because it allows us to deploy new versions of our containers without any downtime, and it helps to ensure that failing containers have little to no effect on our Service.

The Pod resource does not have a replica count, so the behavior is not the same as the Deployment resource. This means that if a Pod manifest (e.g., `kind: Pod`) has a failing container, you, as the developer, will have to intervene manually.

With the provisioned Deployment manifest, you might notice that we *do not* need to change our Service manifest at all. Do you remember why? It's because we used the exact same selector for the Deployment as we did for the Pod. This means that our Service will automatically load balance traffic across all 15 Pods from the Deployment resource (e.g., `my-first-deployment.yaml`) and the one Pod from the Pod resource (`my-first-pod.yaml`), totaling 16 Pods available for our Service resource. To clean up, delete all three resources with the following commands:

```
kubectl delete -f my-first-pod.yaml
kubectl delete -f my-first-deployment.yaml
kubectl delete -f my-first-service.yaml
```

With all of these resources gone, let's see how we can make some modifications to our NGINX container by using a new resource called ConfigMaps.

8.3.4 Customize NGINX with ConfigMaps

ConfigMaps are resources that we can attach to a Deployment (or directly to a Pod resource) as either a volume or as environment variables. ConfigMaps are a great way to modify the default configuration of a container without rebuilding the container image.

In this section, we'll use a ConfigMap resource to modify the default NGINX web page. To do this, create a new folder called `custom-nginx` in the `roadtok8s-kube` directory. Within this folder, create the file `1-configmap.yaml` with the contents of the following listing. Using `1-` will be important, as you will see shortly.

Listing 8.7 K8s YAML manifest for an NGINX ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-configmap
data:
  index.html: |
    <html>
      <head>
        <title>Hello World</title>
      </head>
      <body>
        <h1>Hello World</h1>
      </body>
    </html>
```

The ConfigMap resource uses `data:` instead of `spec:` to define the key-value pairs that we choose to use. In other words, the `data:` blocks are incredibly flexible. What you

see here is we are using the key `index.html` with a multi-line value, denoted by the pipe `|`, that has very rudimentary HTML.

To override the default `index.html` value that comes with the NGINX container image, the ConfigMap key value of `index.html`: in the `data`: will be used in our Deployment. To do this, we will mount the ConfigMap as a *volume* in our Deployment manifest.

When we mount a volume on a Deployment, we are mounting a read-only volume since the ConfigMap is not meant to be modified by the container. If we need to mount read-write volumes, Deployments or Pods can use persistent volumes, which we'll talk about more later in this chapter.

Declaring volumes in a Deployment's template block (or in a pod manifest) is a lot like using volumes with Docker Compose: we must declare the volume and mount the volume. We can have multiple volumes and multiple mounts for each volume. In our case, we will have one volume and one mount.

Create a file called `2-deployment.yaml` within the `custom-nginx` folder in the `road-tok8s-kube` directory with the contents. This Deployment is almost identical to the previous one, with two new additions for declaring the volume (the `volumes`: block) and mounting the volume (the `volumeMounts` within the `containers` definition). The following listing shows this in action.

Listing 8.8 K8s YAML manifest for an NGINX Deployment with a ConfigMap

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: custom-nginx-deploy
spec:
  replicas: 15
  selector:
    matchLabels:
      app: custom-nginx
  template:
    metadata:
      labels:
        app: custom-nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          volumeMounts:
            - name: my-configmap-volume
              mountPath: /usr/share/nginx/html
              readOnly: true
      volumes:
        - name: my-configmap-volume
          configMap:
            name: nginx-configmap
```

Now that we have a Deployment, let's create a new Service for this Deployment. Create a file called `3-service.yaml` within the `custom-nginx` folder in the `roadtok8s-kube` directory with the contents in listing 8.9.

Listing 8.9 K8s YAML manifest for an NGINX Service with a ConfigMap

```
apiVersion: v1
kind: Service
metadata:
  name: custom-nginx-svc
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    nodePort: 30008
  selector:
    app: custom-nginx
```

Notice that the `port` and `nodePort` have changed from our previous Service; this is just to ensure we have no conflicts due to these values. Now our `custom-nginx` folder has the following files:

- `1-configmap.yaml`
- `2-deployment.yaml`
- `3-service.yaml`

The display order within our computer should match the order above. In this case, the provision order is necessary because the Deployment depends on the ConfigMap, and the Service depends on the Deployment. If you have the order done incorrectly, some or all of these Services may not be provisioned correctly or may require them to be provisioned multiple times before it actually does work. We want to have it work on the first go; thus, the order is important. We have the following options to provision these files:

- `kubectl apply -f custom-nginx/1-configmap.yaml`
- `kubectl apply -f custom-nginx/2-deployment.yaml`
- `kubectl apply -f custom-nginx/3-service.yaml`

Or simply:

- `kubectl apply -f custom-nginx/`

The latter will provision all of the files in the folder in the order they are displayed on your computer. This is why I prefixed the files with `1-`, `2-`, and `3-` to ensure the order is correct. Pretty neat feature, huh?

Now if you open the `nodePort` (e.g., `30008`) on any of your node's public IP addresses, you should see the output shown in figure 8.11.



Figure 8.11 ConfigMap-based NGINX container output

Success! You should see the new HTML output instead of the standard NGINX welcome page. If you ran into problems, here are a few things to verify:

- The `nodePort` value you used is not in use by another Service.
- The ConfigMap's name matches the referenced `configMap` block's name in the `volume` block of the Deployment.
- The Deployment volume's name matches the referenced `volumeMounts` block's name in the `containers` block of Deployment.
- You have the correct `index.html` key-value pair in the ConfigMap resource.
- You are not running too many replicas in any given Deployment. Remove any other Deployments and max out around 15 for this example.

We can use ConfigMaps for more than just mounting volumes. We can also use ConfigMaps to set environment variables, as we'll see shortly. Remember, ConfigMaps are not abstracted in any way so you should treat them like you treat any code: leave out all sensitive data.

Before we move on, let's clean up our ConfigMap-based NGINX Deployment with the command `kubectl delete -f custom-nginx/`. Once again, we can shortcut calling all manifests in a folder, in this case for the purpose of deleting the Deployment.

Now that we have seen the volume-mounted use case for ConfigMaps, let's see how we can use ConfigMaps to set environment variables.

8.3.5 *Environment Variables with ConfigMaps and Secrets*

Environment variables are incredibly common to use when deploying containers to any environment. Kubernetes is no different. We can define environment variables in any of the following ways:

- Hard-coded directly in the resource manifest (e.g., in a Pod or Deployment manifest)
- Using a ConfigMap
- Using a Secret

The first option is the least flexible, the least reusable, and probably the least secure. ConfigMaps and Secrets can be attached to an *unlimited* number of resources, thus making them by far the most reusable. With this in mind, changing configuration can be as easy as changing the ConfigMap or Secret instead of changing each individual

resource directly. What's more is that some environment variables can be exposed publicly (even if the public is just teammates) while others cannot be; using a ConfigMap or Secret helps ensure this visibility is minimized. The following listing shows an example of setting environment variables directly in a Deployment manifest.

Listing 8.10 Hard-coded environment variables in a Deployment Manifest

```
kind: Deployment
...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        env:
        - name: MY_ENV_VAR
          value: "This is simple"
        - name: PY_PORT
          value: "8080"
        - name: JS_PORT
          value: "3000"
```

As we can see, this type of setting environment variables is easy, but they are not exactly reusable. What's more, if you accidentally commit this manifest to a public repository with a sensitive environment variable, you could be in a world of trouble. Let's move the `PY_PORT` and `JS_PORT` environment variables to a ConfigMap resource, as seen in the following listing.

Listing 8.11 ConfigMap for environment variables

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: project-ports-cm
data:
  PY_PORT: "8080"
  JS_PORT: "3000"
  SPECIAL: "delivery"
```

Now that we have these ConfigMap variables, we can update our Deployment `env` block like in the following listing.

Listing 8.12 ConfigMap-based environment variables in a Deployment Manifest

```
kind: Deployment
...
spec:
  ...
```

```

template:
  ...
  spec:
    containers:
      - name: nginx
        image: nginx:latest
        env:
          - name: MY_ENV_VAR
            value: "This is simple"
          - name: PY_PORT
            valueFrom:
              configMapKeyRef:
                name: project-ports-cm
                key: PY_PORT
          - name: JS_PORT
            valueFrom:
              configMapKeyRef:
                name: project-ports-cm
                key: JS_PORT
        envFrom:
          - configMapRef:
              name: project-ports-cm

```

What we see here is there are two ways we can use ConfigMaps to set environment variables:

- `valueFrom.configMapKeyRef.name` and `valueFrom.configMapKeyRef.key`—These will grab specific key-value pairs within a specified ConfigMap resource.
- `envFrom.configMapRef.name`—This will load all key-value pairs declared in the `data: block` in the referenced ConfigMap (e.g., `PY_PORT`, `JS_PORT`, and `SPECIAL` from the `projects-port-cm` in listing 8.11).

While ConfigMaps are not secure, their cousin resource, Secrets, are *more* secure. Secrets are meant to hold sensitive data like passwords and API keys, but they are just Base64-encoded values. In other words, they are only slightly more secure than ConfigMaps. That said, let's see how we can use Secrets to set environment variables. To create our Secret resource, create a file called `dont-commit-this-secrets.yaml` in your `roadtok8s-kube` directory with the contents in the following listing.

Listing 8.13 Secret for environment variables

```

apiVersion: v1
kind: Secret
metadata:
  name: somewhat-secret
type: Opaque
stringData:
  SECRET_YT_VIDEO: "dQw4w9WgXcQ"

```

When defining Secrets, we can use two different configurations `data` or `stringData`. The `data` block requires you to submit Base64-encoded values and the `stringData` block requires just raw text. In this case, we are using `stringData` to keep things simple. Now let's verify these Secrets with

- `kubectl -f apply dont-commit-this-secrets.yaml`—Creates the Secret
- `kubectl get secrets`—Lists all Secrets
- `kubectl get secrets somewhat-secret -o yaml`—Lists the specific Secret

After you perform these steps, you should see the same output as in figure 8.12.

```
% kubectl get secrets
NAME                                TYPE                                DATA  AGE
default-token-ssrrg                kubernetes.io/service-account-token 3      2d3h
somewhat-secret                    Opaque                              1      12s
% kubectl get secrets somewhat-secret -o yaml
apiVersion: v1
data:
  SECRET_YT_VIDEO: ZFF3NHc5V2dYY1E=
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Secret","metadata":{"annotations":{},"name":
ingData":{"SECRET_YT_VIDEO":"dQw4w9WgXcQ"},"type":"Opaque"}
      creationTimestamp: "2023-09-02T00:50:09Z"
  name: somewhat-secret
  namespace: default
  resourceVersion: "169650"
  uid: dcce1bed-90db-4d76-863a-f9161e09cf2e
type: Opaque
```

Figure 8.12 `kubectl get secret` output

As you can see, the key `SECRET_YT_VIDEO` is now set to the value `ZFF3NHc5V2dYY1E=`. This is a base64 encoded version of the plain text we submitted. Do a quick search online, and you'll find an easy way to decode this data.

For this reason, I recommend using third-party tools for better securing your true Secret values. A few options are HashiCorp Vault (which has managed and open-source versions), AWS Secrets Manager (a managed service), and Google Secrets Manager (also a managed service).

The best part of Secret and ConfigMap resources is that we can reuse them on any Deployment or Pod within our cluster. While this is yet another potential security risk, it's a great way to remove a lot of redundant information our Kubernetes resources may need.

Now that we have the foundations of a few core Kubernetes resources, let's see how to deploy stateful containers like databases to Kubernetes.

8.4 Volumes and stateful containers

When we used Kubernetes Deployments, you'll notice that each Pod name contains the name of the Deployment along with additional random values (e.g., `custom-nginx-95d4d5bcd-75zn6` or `custom-nginx-95d4d5bcd-4j66r`). These random values are tied to the fact that Pods via Deployments are made to be interchangeable and ephemeral. In other words, the order in which Pods are created or destroyed is not guaranteed. Here's a quick test you can do to see this Pod naming scheme in action:

- 1 Provision a Deployment resource with `kubectl apply -f my-first-deployment.yaml`.
- 2 Run `kubectl get pods`, and you should see a Pod with a name like `custom-nginx-95d4d5bcd-75zn6`.
- 3 Run `kubectl delete pod custom-nginx-95d4d5bcd-75zn6`, and you should see a new Pod created with a new name like `custom-nginx-95d4d5bcd-4j66r`.
- 4 Rinse and repeat. Deployments will always create new Pods with new names and aim to reach the replica count.

The Deployment resource's Pod naming scheme means using specific Pods by name is a bit finicky. For example, if we wanted to use the *bash shell* of a running Pod within a Deployment, we have two options:

- `kubectl exec -it <pod-name> -- bin/bash`—This, of course, requires we know the `<pod-name>`, which is nearly impossible to predict, so we can use `kubectl get pods -l app=some-label` to narrow down the group of Pods for any given Deployment. This filter at least allows us to find a valid `<pod-name>`. But remember, this newly found `<pod-name>` is almost certainly going to *change*, especially if we make *any* changes to the Deployment or if the Pod needs to be updated otherwise.
- `kubectl exec -it deployments/<deployment-name> -- bin/bash`—This command is what I recommend using if you do not need to diagnose problems with a specific Pod (because sometimes you might). Using `deployments/<deployment-name>`, where `<deployment-name>` is the `metadata.name` value for a Deployment, will look for a Pod within the group of Pods managed by the Deployment.

The format of `kubectl <command> deployments/<deployment-name>` works great for the following:

- *Logs*—`kubectl logs <pod-name>` or `kubectl logs deployments/<deployment-name>`. Use this command to review logs related to a Pod or Deployment.
- *Describe*—`kubectl describe pod <pod-name>` or `kubectl describe deployments/<deployment-name>`. Use this command to better understand the details of a Pod or a pod managed by a Deployment.

This strange naming scheme was not just a sneaky way to introduce a few new commands; it was also to help us ask the question: Can we have logical Pod names? Something like `custom-nginx-01` or `custom-nginx-02` would be nice. While we could create a Pod manifest with these auto-incrementing names (e.g., `kind: Pod`), that would be wasteful and downright silly when we have a Kubernetes resource that does this already: the `StatefulSet` resource.

Are `StatefulSets` just `Deployments` that name Pods differently? Yes and no. While `StatefulSets` do name Pods in order (e.g., `custom-nginx-01` or `custom-nginx-02`), they also have a major other benefit: *volume claim templates*.

Before we provision our first `StatefulSet`, let's look at volume claims within a `Deployment`. Volume claims are requests to the cloud provider for access to a managed storage volume that can be attached to a Pod.

8.4.1 Volumes and Deployments

Persistent volumes, or Block Storage volumes, can be added to a cluster via a `PersistentVolumeClaim` (PVC) resource. With a cloud-managed Kubernetes cluster, defining this manifest resource will ping your cloud provider (in our case, Akamai Connected Cloud) and request a new volume. This volume will be attached to a node in your cluster and will be available for use in your cluster.

Create a new directory called `deploy-pvc`, in which we will do the following:

- Create a `PersistentVolumeClaim` resource.
- Create a `Deployment` resource.

If successful, the `PersistentVolumeClaim` resource will provide us with a new volume that we can attach to our `Deployment` resource. This volume will be attached to a node in our cluster and will be available for use in our cluster. Here are the primary configurations we need to define in our `PersistentVolumeClaim` resource:

- `accessModes`—Kubernetes offers a wide range of access modes, including `ReadWriteOnce`, `ReadOnlyMany`, and `ReadWriteMany`. Even with this, not all cloud providers support all access modes. For example, Akamai Connected Cloud only supports `ReadWriteOnce`, which means the volume cannot be shared across multiple Pods.
- `resources.requests.storage`—This is the size of the volume we want to request. This value is in gigabytes: `10Gi` is 10 gigabytes of storage.
- `storageClassName`—While there are numerous options for this, we will use a cloud-provided value. In our case, we can use `linode-block-storage` for drives that should delete along with the PVC resource or `linode-block-storage-retain` for drives that will remain in your account even if the PVC resource is deleted.

Create a file called `1-pvc.yaml` in the `deploy-pvc` directory with the contents in the following listing.

Listing 8.14 PersistentVolumeClaim resource

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: first-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: linode-block-storage-retain

```

With this PVC resource, we can now attach this volume to a Deployment resource. Create a file called `2-deployment.yaml` in the `deploy-pvc` directory with the contents in the following listing.

Listing 8.15 Deployment resource with a PVC

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  ...
  template:
    ...
    spec:
      ...
      containers:
        - name: nginx
          ...
          volumeMounts:
            - mountPath: "/usr/share/nginx/html"
              name: first-pv-storage
      volumes:
        - name: first-pv-storage
          persistentVolumeClaim:
            claimName: first-pvc

```

Apply these resources with `kubectl apply -f deploy-pvc/`. After a short wait, the following should happen:

- Only a single `nginx-deployment` Pod (e.g., `replicas: 1`) should be running.
- A new volume should be provisioned. To verify, run `kubectl describe pvc first-pvc`, look for the `Volume: <value>` and it should resemble `pvc-b50fb2b7b4744cce`. This is the exact label for the volume in your Akamai Connected Cloud account.

Now that we have seen our first volume generated, let's discuss limitations to the `ReadWriteOnce` access mode: the replica limit. By default, this particular access mode means the volume can only be attached to one Pod at any given time. In other words, this volume cannot be shared across multiple Pods. If we scale our Deployment to two or more replicas, the new Pods will not provision because the volume is already attached to the first Pod. This is a limitation of the `ReadWriteOnce` access mode and not a limitation of Kubernetes.

As you may know, multiple computers or apps or Pods sharing the same persistent volume can often lead to data corruption (i.e., may different processes writing to the exact same file at the same time). What's more, there are better storage options (e.g., databases, message queues, s3-compatible object storage, etc.) for sharing data across multiple Pods. I believe it's for this reason that many cloud providers, Akamai Connected Cloud included, opt for only supporting `ReadWriteOnce` access mode at the time of writing this book.

Before we move on, destroy these resources with `kubectl delete -f deploy-pvc/`; this will delete both the PVC, the volume in your Akamai account, and the Deployment resources.

While you may still need multiple containers running with access to storage volumes, you can do so by using another Kubernetes resource: the `StatefulSet`. Let's see how we can attach a volume to each Pod in a `StatefulSet`.

8.4.2 StatefulSets

Throughout this book, we have seen the use of “Hello World” examples with our web applications. While these apps have been useful in explaining a lot of concepts, they lack a key layer of practicality. While web apps *can* run independently of data stores (e.g., databases, message queues, and so on), as we have done many times in this book, the real valuable web apps rarely do. Data stores are a combination of an application running and a storage volume that application uses to read and write data. This means that the data store application itself is *stateless* and can be a container, while the data itself is *stateful* and needs to be in a storage volume. `StatefulSets` provide a simple way to attach a storage volume to each Pod replica declared by the manifest, thus ensuring the Pods are *stateful* and perfect candidates for running data stores.

`StatefulSets` unlock the ability to run stateful Pods (containers) by providing a few key features:

- *Volume template*—Every new Pod in a `StatefulSet` will have a new volume attached to it.
- *Pod naming scheme*—Every new Pod in a `StatefulSet` will have a new name that is predictable and ordered.
- *Pod ordering*—Every new Pod in a `StatefulSet` will be created in order. This means that `custom-nginx-01` will be created before `custom-nginx-02` and so on.
- *Pod deletion*—Every new Pod in a `StatefulSet` will be deleted in reverse order. This means that `custom-nginx-03` will be deleted before `custom-nginx-02` and so on.

These features make StatefulSets an excellent way to deploy stateful containers like databases, message queues, and other applications that require a persistent volume. Keep in mind that deploying multiple instances of any given container does not automatically connect the instances to each other. In other words, deploying a Postgres database with a StatefulSet does not automatically create a Postgres cluster. Creating clustered databases is a topic for another book.

To create a StatefulSet, we need to define a few key configurations:

- `initContainers`:—When we mount a new volume to a container, it comes with a directory called `lost+found`. This directory must be removed for the Postgres container to work properly. The `initContainers` block allows us to run a container before the main container is started. In our case, we'll run a minimal container image that mounts our volume, removes the `lost+found` directory with the command `sh -c "rm -rf lost+found"`, and then exits. While the `lost+found` folder can be used for file recovery, we have no files to recover at this time. This is a common pattern for initializing volumes and is a great way to ensure that our Postgres container will work properly without accidentally deleting the data we actually want.
- `replicas`:—The number of Pods to run; defaults to 1.
- `selector`:—Just as with Deployments, our StatefulSet needs to know which Pods to manage through the selector.
- `template`:—The configuration we use here is identical to a standalone Pod manifest without the `kind` and `apiVersion` declaration. It's called a template because it contains the default values for the group of Pods the StatefulSet will manage.
- `volumeClaimTemplates`:—When creating new Pods, this configuration includes the volume claim template that will be used for each Pod. The format is identical to the PVC resource we used in the previous section without the `kind` and `apiVersion` declarations.

With this in mind, let's create a new folder called `postgres-db` with the following files:

- `1-secrets.yaml`—Be sure to add this to `.gitignore` as it contains sensitive data. This file will be used to configure the default Postgres database name, user, and password.
- `2-statefulset.yaml`—This is the StatefulSet resource that will create our Postgres Pods and PVCs for each Pod.

Create a file called `1-secrets.yaml` in the `postgres-db` directory with the contents in the following listing.

Listing 8.16 Secrets for Postgres

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-db-secret
type: Opaque
```



```
stringData:
  DB_NAME: "postgres-db"
  DB_USER: "postgres"
  DB_PASSWORD: "dQw4w9WgXcQ"
```

Next, create a file called `2-statefulset.yaml` in the `postgres-db` directory and start with the contents in the following listing.

Listing 8.17 StatefulSet for Postgres, part 1

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
```

At this point, this is identical to what we might see in a Deployment resource, except we are using `kind: Service`. Before we add the Pod template configuration, let's add the `volumeClaimTemplates: block`. The `spec.template` and `spec.volumeClaimTemplates` blocks need to be on the same level as seen in the following listing.

Listing 8.18 StatefulSet for Postgres, part 2

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      ...
    spec:
      ...
  volumeClaimTemplates:
  - metadata:
      name: db-vol
    spec:
      storageClassName: linode-block-storage-retain
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 20Gi
```

Now let's start the process of defining our Pod template within the StatefulSet. First, we need the metadata label to match the selector, and then we need to define the `initContainers` block, as seen in the following listing.

Listing 8.19 StatefulSet for Postgres, part 3

```
...
spec:
  ...
  template:
    metadata:
      labels:
        app: postgres
    spec:
      initContainers:
        - name: delete-lost-found
          image: alpine:latest
          command: ["sh", "-c", "rm -rf /mnt/lost+found"]
          volumeMounts:
            - name: db-vol
              mountPath: /mnt
      volumeClaimTemplates:
        - metadata:
            name: db-vol
          spec:
            ...
```

Now we can define the `containers:` block.

Listing 8.20 StatefulSet for Postgres, part 4

```
...
spec:
  ...
  template:
    metadata:
      containers:
        - name: postgres-container
          image: postgres:12.16
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 5432
          env:
            - name: POSTGRES_DB
              valueFrom:
                secretKeyRef:
                  name: postgres-db-secret
                  key: DB_NAME
            - name: POSTGRES_USER
              valueFrom:
                secretKeyRef:
                  name: postgres-db-secret
                  key: DB_USER
            - name: POSTGRES_PASSWORD
```

```

    valueFrom:
      secretKeyRef:
        name: postgres-db-secret
        key: DB_PASSWORD
  volumeMounts:
  - name: db-vol
    mountPath: /var/lib/postgresql
  volumeClaimTemplates:
  - metadata:
      name: db-vol
    spec:
      ...

```

This StatefulSet is a very practical example that combines a lot of what we have learned so far. Here’s a quick summary of what we have defined:

- A single replica (e.g., `replicas: 1`).
- A selector that matches the label `app: postgres`.
- A template that matches the label `app: postgres` and has a container with the image `postgres:12.16`.
- Environment variables `POSTGRES_DB`, `POSTGRES_USER`, and `POSTGRES_PASSWORD` are set to the related `DB_` values in the `postgres-secrets` resource.
- An init container that mounts the volume and removes the `lost+found` directory.
- A volume claim template that requests a 20 GB volume with the `ReadWriteOnce` access mode.

Before we verify this StatefulSet, let’s create a Service to allow access to the StatefulSet’s provision Pods via the defined selector. This time, we’ll use `ClusterIP` so we don’t expose our database to the internet. Create a file called `3-service.yaml` in the `postgres-db` directory with the contents in the following listing.

Listing 8.21 ClusterIP Service for Postgres

```

apiVersion: v1
kind: Service
metadata:
  name: postgres-svc
spec:
  type: ClusterIP
  ports:
  - port: 5432
    targetPort: 5432
  selector:
    app: postgres

```

With these manifests defined, run `kubectl apply -f postgres-db/`. After a short duration, the database should be provisioned and running at the Pod `postgres-0`.

To verify the PostgreSQL database is working, we can use the port forwarding feature within `kubectl`. This feature enables our local computer, as in `localhost`, to connect to a cluster's resource. Here are a few ways:

- Port forward a Pod—`kubectl port-forward <pod-name> <local-port>:<k8s-container-port>` like `kubectl port-forward postgres-0 3312:5432` and then on your local computer, use `localhost:3312` along with your favorite PostgreSQL client.
- Port forward a StatefulSet—`kubectl port-forward statefulsets/postgres 3312:5432`. This will default to the first Pod in the StatefulSet.
- Port forward a Service—`kubectl port-forward service/postgres 3312:5432`.

If you have the `psql` PostgreSQL client installed on your local machine, you can run `PG_PASSWORD=<"password-set"> psql -h localhost -U postgres -d <db-set> -p <port-set>` like `PG_PASSWORD="dQw4w9WgXcQ" psql -h localhost -U postgres -d postgres-db -p 3312`.

If you do not have the `psql` client installed, just run `kubectl get pods` (to get a list of running Pods) and `kubectl get svc` (to get a list of Services). Both of these commands should verify the Pods are working. If they are not running, use `kubectl describe pod <pod-name>` to see what is going on.

While using a local version of the `psql` client works well, it doesn't show us how to do container-to-container communication from within our cluster. Let's do that now.

8.4.3 *Container-to-container communication within Kubernetes*

While I think updating our Python or Node.js applications with a database would be a good idea, it would take far too much code to do so, and we would lose sight of the important Kubernetes bits. Instead, we'll use some great Kubernetes features to connect to our database from within our cluster by provisioning a temporary and one-off Pod, install the PostgreSQL client, and then connect to our database.

Creating a container every time you need to test something is not always practical. In our case, we want to test the `psql` command from within our cluster, but we do not have a container with the `psql` command installed. While this exact scenario might not happen to you often, it's a good use case to *skip building the container* and instead use a *deletable Pod* we can install things on.

To do this, we will use manual provisioning of a Pod directly in the command line with `kubectl`. In the long run, I always recommend using manifests (YAML) for provisioning resources, but knowing how to create a one-off Pod with `kubectl` can help you test various configurations (i.e., with ConfigMaps or Secrets), container images, and container-to-container communication in your cluster. In many ways, creating one-off Pods with `kubectl` is a lot like creating one-off Pods with the Docker command-line interface.

To run a deletable container with just Docker, we run the command: `docker run --rm <container-image-name> --name <runtime-name> -p 8080:8080` where `<container-image-name>` is the name of the container image, `<runtime-name>` is the name

of the container, and `-p 8080:8080` is the port mapping. The deletable part is the `--rm` flag, which tells Docker to delete the container when it exits.

To run a delete-at-anytime Pod with `kubectl`, we will use the following:

- A `debian:bullseye-slim` container image because it is about 30 mb and thus quick to provision.
- Debian has the `apt` package manager like Ubuntu; it just has a smaller footprint. Ubuntu is a Debian system, just with many more built-in tools and, thus, a larger footprint. Since we have been using Ubuntu throughout this book, these commands should be familiar.
- `sleep infinity` is an entrypoint command we can use to keep the Pod's container running until we delete the Pod. If we did not include the `sleep infinity`, the Pod would exit immediately after starting because there is no command or entry point to run.

With this in mind, let's run the following commands:

- 1 `kubectl run my-psql --image=debian:bullseye-slim -- sleep infinity`. After running this command, you can verify the Pod with `kubectl get pods my-psql -o yaml`. Using the `-o yaml` will output the current manifest of this Pod including the state in the K8s cluster.
- 2 Enter the Pod's shell with `kubectl exec -it my-psql -- /bin/sh`. This will give us the ability to run commands within the Pod's container and communicate with other Pods within the cluster.
- 3 Install the `psql` client with `apt update && apt install -y postgresql-client`. You do not need the `sudo` command because of how the `debian:bullseye-slim` container image is configured.

At this point, you have a manually created Pod running a container that has `psql` installed. Let's connect this `psql` client to our PostgreSQL database Pod via the `postgres-svc`. To do so, we have two options:

- The cluster IP for our `postgres-svc` Service
- The Kubernetes' built-in DNS Service for the `postgres-svc` Service

Before we see the built-in DNS Service, let's get the value of the cluster IP for this Service. It's as simple as running `kubectl get service <service-name>` like `kubectl get service postgres-svc`. The output should resemble figure 8.13.

```

$ kubectl get service postgres-svc
NAME         TYPE          CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
postgres-svc ClusterIP      10.128.189.63 <none>       5432/TCP    159m

```

Figure 8.13 `kubectl get service` output

In the Cluster-IP column in figure 8.13, the value is 10.128.189.63. This value is the internal IP address of this particular Service. I would be willing to bet you have a different value, but even if you didn't, you absolutely could. The Cluster IP value is the host value we can use to connect to this Service from within our cluster. As a reminder, Kubernetes Service resources do not care what they are attached to; the cluster IP will attempt to forward the requests to the correct Pod as you have it configured.

While the cluster IP is a valid approach, it's flawed for a major reason: the given IP address is unpredictable. An unpredictable IP means how our Pods communicate with each other becomes a game of constantly checking the provisioned internal dynamically provisioned IP address values. What's better is we can use Kubernetes' built-in DNS Service to connect to our database using very predictable values. The format is as follows:

```
<resource-name>.<namespace>.<resource-type>.cluster.local.
```

For our example, it breaks down like the following:

- `<resource-name>`—Our resource name is `postgres-svc`. The resource type, in this case, is `Service`, but that doesn't apply here.
- `<namespace>`—The default Namespace is literally `default`. We'll see how to change this value shortly.
- `<resource-type>`—Since we are using a `Service`, the `<resource-type>` value is `svc`, much like `kubectl get svc` will display all `Services`.
- `cluster.local`—This is the default DNS suffix for all resources in Kubernetes. While this can be modified, it's a more advanced topic.

Here are a few examples of DNS endpoints for various `Services` in various `Namespaces` (more on `Namespaces` shortly):

- `postgres-svc.default.svc.cluster.local` is the DNS endpoint for the `postgres-svc` `Service` in the `default` `Namespace`.
- `redis-svc.default.svc.cluster.local` would be the DNS endpoint for a `Service` with the name `redis-svc` in the `default` `Namespace`.
- `mysql.doubleunit.svc.cluster.local` would be the DNS endpoint for a `Service` with the name `mysql` in the `doubleunit` `Namespace`.
- `mariadb.imf.svc.cluster.local` would be the DNS endpoint for a `Service` with the name `mariadb` in the `imf` `Namespace`.

I'll let you decide which is easier for you, the DNS endpoint or the cluster IP. Now let's put both commands to the test, starting with the correct DNS endpoint. Be sure to enter the bash shell with `kubectl exec -it my-psql -- /bin/sh`. Then run the following:

- `psql -h postgres-svc.default.svc.cluster.local -p 5432 -U postgres -d postgres-db`—This should prompt you for a password. The username (`-U`), the database name (`-d`), and the password (prompted) were all set in the `postgres-secrets` resource. The port (`-p`) is the default port for PostgreSQL as well as the port we set in the `postgres-svc` `Service`.

- 1 Type the SQL command: `CREATE DATABASE hello_world;`
- 2 Exit the `psql` client with `\q`.
- 3 Now using the Cluster IP, `psql -h 10.128.189.63 -p 5432 -U postgres -d postgres-db`. The only change here is the hostname (`-h`).
- 4 List the tables with `\dt` and you should see the `hello_world` database listed.
- 5 Exit the `psql` client with `\q`.
- 6 Exit the bash shell with `exit`.

Each of these steps is encapsulated in figure 8.14.

```

% kubectl exec -it my-psql -- /bin/sh
# psql -h postgres-svc.default.svc.cluster.local -p 5432 -U postgres -d postgres-db
Password for user postgres:
psql (13.11 (Debian 13.11-0+deb11u1), server 15.4 (Debian 15.4-1.pgdg120+1))
WARNING: psql major version 13, server major version 15.
        Some psql features might not work.
Type "help" for help.

postgres-db=# CREATE DATABASE hello_world
postgres-db=# \q
# psql -h 10.128.12.138 -p 5432 -U postgres -d postgres-db
Password for user postgres:
psql (13.11 (Debian 13.11-0+deb11u1), server 15.4 (Debian 15.4-1.pgdg120+1))
WARNING: psql major version 13, server major version 15.
        Some psql features might not work.
Type "help" for help.

postgres-db=# \l

```

List of databases					
Name	Owner	Encoding	Collate	Ctype	Access privileges
postgres	postgres	UTF8	en_US.utf8	en_US.utf8	
postgres-db	postgres	UTF8	en_US.utf8	en_US.utf8	
template0	postgres	UTF8	en_US.utf8	en_US.utf8	=c/postgres +
template1	postgres	UTF8	en_US.utf8	en_US.utf8	postgres=CTc/postgres +
					=c/postgres +
					postgres=CTc/postgres

```

(4 rows)

postgres-db=# \q
# exit

```

Figure 8.14 Cross-service communication between Services and `psql`

While this is about as basic as it gets for SQL commands with `psql` and a PostgreSQL database, it's a great example of how to connect to a database from within a Kubernetes cluster and, more importantly, do container-to-container communication.

8.4.4 Namespaces to manage cluster resources

In this section, we'll take a quick look at how to use *Namespaces* to group resources in a different way. Namespaces can be applied to any Kubernetes resource and are a great way to group resources by team, project, or even environment.

If a Namespace is *not used* or declared, all resources will be provisioned in the automatically added Kubernetes Namespace called `default`. You can verify that the default Namespace exists by running `kubectl get namespaces` to review all available Namespaces. In other words, when we ran `kubectl apply -f ...`, we were actually running something like `kubectl apply -f ... -n default`. The `-n` flag is used to specify a Namespace (unless it's declared in the manifest, as we'll learn about shortly).

Since we just saw how Namespaces can affect Kubernetes-managed DNS endpoints from within a cluster, let's create another example data store using Redis. Redis is another stateful application that requires a StatefulSet, much like PostgreSQL before. In this section, we use the Namespace resource to highlight how we can further isolate and distinguish various Pods and Services within our cluster. Of course, there are some technical differences between Redis and PostgreSQL that are outside the scope of this book, but I think it's important to see how Namespaces can help further divide these two pieces of software.

Create a new folder called `redis-db` with the following files:

- *1-namespace.yaml*—The Namespace resource that will create our `redis-db` Namespace.
- *2-statefulset.yaml*—The StatefulSet resource that will create our Redis Pods and PVCs for each Pod.

While Redis can be protected through a password, we will focus more on the Namespace aspect of this example. Create a file called `1-namespace.yaml` in the `redis-db` directory.

Listing 8.22 Namespace resource for Redis

```
apiVersion: v1
kind: Namespace
metadata:
  name: rk8s-redis
```

Before we create anything else, provision the Namespace with `kubectl apply -f redis-db/1-namespace.yaml`. Once complete, run `kubectl get pods -n rk8s-redis` and then `kubectl get pods -n default`. If you still have your Postgres StatefulSet running, you should see Pods in the second `get pods` command but not the first. This is because the `-n` flag is used to filter resources by Namespace.

While this feels similar to our selector filter, it's different. While Namespaces can be used to filter results with a selector, we can also create specific user accounts to limit access to specific Namespaces and resources within those Namespaces. We can also configure policies to prevent internal communication across Namespaces. This kind of configuration is outside the scope of this book but good to know about.

So how do we add resources to a Namespace? Two ways:

- Add the `metadata.namespace` value to the resource manifest. This is the most common way to add a resource to a Namespace.

- Use the `-n` flag with `kubectl apply -f <resource-manifest> -n <namespace>` like `kubectl apply -f postgres-db/ -n rk8s-redis`. The `-n` flag with `kubectl apply` will not work if the manifest has a `metadata.namespace` already defined.

With this in mind, create the file `2-statefulset.yaml` in the `redis-db` folder for our Redis StatefulSet.

Listing 8.23 StatefulSet for Redis

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
  namespace: rk8s-redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:latest
          ports:
            - containerPort: 6379
          volumeMounts:
            - name: redis-data
              mountPath: /data
      volumeClaimTemplates:
        - metadata:
            name: redis-data
          spec:
            storageClassName: linode-block-storage
            accessModes: [ "ReadWriteOnce" ]
            resources:
              requests:
                storage: 10Gi
```

As we can see, Redis does not require an `InitContainers` declaration like PostgreSQL did. Now let's create a Service for this StatefulSet. Create a file called `3-service.yaml` in the `redis-db` directory.

Listing 8.24 ClusterIP Service for Redis

```
apiVersion: v1
kind: Service
metadata:
  name: redis-svc
  namespace: rk8s-redis
```

```
spec:
  type: ClusterIP
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
```

Apply the changes with `kubectl apply -f redis-db/`. After a short duration, the database should be provisioned and running at the Pod `redis-0`. Now ask yourself, what is the DNS endpoint for this Service? If you said `redis-svc.rk8s-redis.svc.cluster.local`, you are correct. Let's verify this by following a similar pattern to the one we used to verify the PostgreSQL database, StatefulSet, and Service:

- 1 Enter the temporary Pod's shell with `kubectl exec -it my-psql - /bin/sh`.
- 2 Install the Redis client library with `apt update && apt install -y redis-tools`.
- 3 Verify the Redis client exists with the command `-v redis-cli`, which should return `/usr/bin/redis-cli`.
- 4 Use the `redis-cli ping` command with the host `-h` flag as: `redis-cli -h redis-svc.rk8s-redis.svc.cluster.local ping`. This should return `PONG`, which means the connection was successful.

We now have a working Redis database that we can connect to from within our cluster. What's more, we can communicate via internal DNS from Pod-to-Pod via a Service resource regardless of the Namespace.

If you are done with the temporary Pod, feel free to delete it now with `kubectl delete pod my-psql`. You also might consider removing your PostgreSQL and Redis resources with `kubectl delete -f postgres-db/` and `kubectl delete -f redis-db/`, respectively. As with many cloud-managed resources, volumes *are* going to be charged extra; while this number is not going to be high, it's good to know and delete resources you are not going to use.

Now that we thoroughly understand how to deploy containers, both stateless and stateful, let's put our Python and Node.js applications to work on Kubernetes along with a custom domain name.

8.5 *Deploy apps to production with Kubernetes*

In this section, we'll deploy our stateless and containerized applications we have been working on throughout the book. Given the apps are stateless, I will default to using a Kubernetes Deployment resource. To deploy our applications to production, we will do the following:

- Create ConfigMaps for environment variables with each app.
- Define Deployment manifests to deploy the running containers.
- Implement a load-balancing service to route external traffic to our applications.

Routing external, or internet-based, traffic to our applications is critical for any production application. To route incoming traffic, also known as ingress traffic, we have two built-in options with the Kubernetes Service resource: NodePort or LoadBalancer. As we discussed previously, using a NodePort Service will use a port on each node, and if the configuration is left blank, this port will be automatically assigned to a non-standard HTTP port value to make room for other Services that use NodePort. If you intend to have a custom domain, you need a standard HTTP port of 80 or 443 for internet traffic, as web browsers default to Port 80 for HTTP traffic and Port 443 for HTTPS traffic. We will use Port 80 since Port 443 is for secure traffic but requires additional configuration and certificates that fall outside the scope of this book. Using a standard HTTP port (80 or 443) means we can use a custom domain (e.g., `roadtok8s.com`) without specifying a port number (e.g., `roadtok8s.com:30000`) like you might with a NodePort Service.

Since we want to use Port 80, we still *could* use NodePort configured to Port 80, but that would mean that only one Service can be on Port 80 since only one Service can run on a single port. This is not a problem if you only have one service, but if you have two or more Services, you need to use a LoadBalancer Service. Let's take a look.

8.5.1 LoadBalancer Services

All Kubernetes Service resources perform load balancing regardless of the defined type. This means that if a request comes in, Kubernetes will route the request to the correct Pod and consider the amount of traffic, or load, that Pod is currently dealing with. The ClusterIP service resource performs internal load balancing across Pods. The NodePort exposes a node's port to again load balance requests to Pods *across* the entire cluster (regardless of which node the request comes from). If we want to load balance across the virtual machines (including their current status) *and* across Pods within the cluster, we can use the LoadBalancer Service type.

With a managed Kubernetes cluster, using the LoadBalancer Service type will request a managed load balancer from the cloud provider. In our case, using this type means we will request a NodeBalancer directly from Akamai Connected Cloud. The Linode NodeBalancer is a managed load balancer service that provides a static public IP address along with load balancing requests to our cluster's nodes. Managed load balancers like the Linode NodeBalancer often have additional costs associated with them: it's typically not a free service.

In managed load-balancer environments, such as NodeBalance in Akamai Connected cloud, a static public IP address is usually provided. This static IP can serve as the target destination for our custom domain by configuring an address record (*A record*). An *A record* is a DNS setting within our domain hosting service (e.g., Route53, Linode Domains, Name.com, GoDaddy.com, etc.) that maps a domain name to an IP address. By updating this record, we can direct custom domains like `py.book.roadtok8s.com` or `js.book.roadtok8s.com` to the static public IP of our load balancer. This makes it straightforward to associate custom domains with Services in a Kubernetes cluster.

Before we create our Services, create the folder `apps` in the `roadtok8s-kube` directory for both of our applications (Python and Node.js) and add the following as empty placeholder files:

- `1-configmap.yaml`
- `2-deployment-py.yaml`
- `3-deployment-js.yaml`
- `4-service-py.yaml`
- `5-service-js.yaml`

Before we look at the ConfigMap, let's think about the internal domain name we want for each application's Service. Here's what I think would be good:

- *Python*—`py-book.default.svc.cluster.local`
- *Node.js*—`js-book.default.svc.cluster.local`

The reason I am defining these now is because they will be our north star for the ConfigMap and the Service for each app. With this in mind, let's create a ConfigMap for each application. Create a file called `1-configmap.yaml` in the `apps` directory.

Listing 8.25 ConfigMap for apps

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: rk8s-apps
data:
  PY_ENDPOINT: "py-book.default.svc.cluster.local"
  PY_PORT: "8080"
  JS_ENDPOINT: "js-book.default.svc.cluster.local"
  JS_PORT: "3000"
```

Each of these values will be used in the Deployment resources for each application, which means we need this ConfigMap on our cluster before the Deployment resources are created (thus the `1-` prefix).

While `PY_ENDPOINT` and `JS_ENDPOINT` are not going to be used in this book, they exist to show more examples of DNS endpoints in case you decide to modify your web applications to communicate with each other in the cluster.

With this in mind, let's create the Deployment resources for each application. Create a file called `2-deployment-py.yaml` in the `apps` directory.

Listing 8.26 Deployment for Python app

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: py-deploy
spec:
```

```

replicas: 3
selector:
  matchLabels:
    app: py-deploy
template:
  metadata:
    labels:
      app: py-deploy
  spec:
    containers:
      - name: rk8s-py
        image: codingforentrepreneurs/rk8s-py:latest
        ports:
          - name: http-port
            containerPort: 8080
        env:
          - name: PORT
            valueFrom:
              configMapKeyRef:
                name: rk8s-apps
                key: PY_PORT
        envFrom:
          - configMapRef:
              name: rk8s-apps

```

The only new addition is that we named the container port `http-port`. Naming the port makes it more reusable when we provision the service. While this is an optional step, it's very convenient when you are doing multiple Deployments with multiple container ports the way we are.

To create the Node.js version of this Deployment, copy this file, rename it to `3-deployment-js.yaml`, and update the following values:

- Change the `containerPort` value to match the Node.js application's port. In my case, I will change it to 3000.
- Replace all instances of `py-` with `js-` so the name and the selector is `js-deploy` and `PY_` with `JS_` so we are using `JS_PORT` for the environment variable `PORT`.
- Set the container image to `codingforentrepreneurs/rk8s-js:latest` or the image you have on Docker Hub.
- Set the `PORT` value to 3000.

At this point, you can run `kubectl apply -f apps/` and verify that the Pods are running for both container images. If there are errors, consider using `nginx` as the container image to verify the container you're using is not the problem. As a reminder, you can use the following commands to verify the Pods are running:

- `kubectl get pods -l app=py-deploy` or `kubectl get pods -l app=js-deploy`
- `kubectl describe deployments/py-deploy` or `kubectl describe deployments/js-deploy`
- `kubectl logs deployments/py-deploy` or `kubectl logs deployments/js-deploy`

With the Deployments in place, let's create the Service resources for each application. Create a file called `4-service-py.yaml` in the `apps` directory.

Listing 8.27 Service for the Python app

```
apiVersion: v1
kind: Service
metadata:
  name: py-book
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: http-port
  selector:
    app: py-deploy
```

We want to set the port to 80 because this is the default HTTP port value, which means our application will be available at `http://<loadblanancer-static-ip>:80` and `http://<loadblanancer-static-ip>;`.

Once again, copy this file, rename it to `5-service-js.yaml`, and update the following values:

- Change the Service name from `py-book` to `js-book` (remember the DNS names?).
- Change the selector from `py-deploy` to `js-deploy`.

Now run `kubectl apply -f apps/` and verify the Pods and Services were created with `kubectl get pods` and `kubectl get svc` (or `kubectl get services`), which should result much like figure 8.15.

```
% kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
js-deploy-86b77bf6d7-h5n2s         1/1     Running   0           3m18s
js-deploy-86b77bf6d7-qr4cm         1/1     Running   0           3m22s
js-deploy-86b77bf6d7-wzht6         1/1     Running   0           3m19s
my-psql                             1/1     Running   0           23h
py-deploy-648989c4cb-7k7jr         1/1     Running   0           3m22s
py-deploy-648989c4cb-cd4fx         1/1     Running   0           3m18s
py-deploy-648989c4cb-p6dw4         1/1     Running   0           3m20s
% kubectl get svc
NAME      TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
js-book   LoadBalancer 10.128.253.219   172.232.176.126  80:30890/TCP     16s
kubernetes ClusterIP    10.128.0.1       <none>           443/TCP          5d19h
py-book   LoadBalancer 10.128.62.33    172.232.176.29   80:32000/TCP     3m26s
```

Figure 8.15 `kubectl get pods` and `Services` output

Both of your Services should have external IP addresses that will direct traffic to your Deployments via the LoadBalancer Service. If you do not see an external IP address, wait a few minutes, as sometimes the LoadBalancer takes time to provision a new static public IP address. If you still do not see an external IP address, use `kubectl describe service py-book` or `kubectl describe service js-book` to see what is going on.

Now that you have a public static IP address, you can point your domain name to it. To do so, you must use an A Record with the value being the IP address of the LoadBalancer Service and where the subdomain can be anything you choose.

8.5.2 Deploying to Kubernetes with GitHub Actions

Now it's time to automate Kubernetes Deployments with CI/CD workflows on GitHub Actions. As before, the workflow you will design here can be modified to work with other CI/CD platforms with some minor tweaking for one simple reason: *all the tools are open-source*. The key to automating a Kubernetes Deployment is

- A valid service account with the correct permissions
- A valid kubeconfig file for the service account
- A valid `kubectl` version
- A running Kubernetes Cluster

Creating and managing service accounts is a more advanced security topic that is outside the scope of this book. If you are interested, you can read my blog post (<https://www.codingforentrepreneurs.com/blog/kubernetes-rbac-service-account-github-actions/>) because it relates to GitHub Actions and Kubernetes with Akamai Connected Cloud's Linode Kubernetes Engine.

While this process *can* happen in the same GitHub repositories as our Python and Node.js applications, I recommend separating the infrastructure (Kubernetes) from the application. Our applications can still build the containers, but they do not need to be responsible for configuring the Kubernetes cluster.

Our workflow will do the following:

- Add our kubeconfig file to the default `kubectl` location: `~/.kube/config`.
- Echo running Pods and Deployments on our *default* Namespace with `kubectl get pods` and `kubectl get deployments`.
- Apply any manifest changes in the `apps/` folder with `kubectl apply -f apps/`.
- Apply a rollout restart to the `py-deploy` and `js-deploy` Deployments with `kubectl rollout restart deployment py-deploy` and `kubectl rollout restart deployment js-deploy`. The `kubectl rollout` command is a great way to update Deployments without having to delete and recreate them without runtimes.

As you may have noticed in the Deployment manifests (e.g., `apps/2-deployment-py.yaml`), we used the `:latest` tag for each image. If we run `kubectl apply -f apps/` again with the same manifest, Kubernetes will not know to update the Deployment because the manifest has not changed. If the manifest does change, then Kubernetes knows to start a new Deployment rollout.

Explicitly setting the Deployment's container tag (e.g., `:latest`) to a version number or commit ID, is considered the best practice for production deployments for the following reasons:

- *Historical record*—Easier auditing and rollback right in your git repo history.
- *Debugging*—Quickly find problems with any given container.
- *Deployment failures*—If the deployment fails altogether in an automated release, it will be much easier to identify and roll back faulty releases.
- *Immutability*—If done correctly, each tag will be a unique, unchanging version.
- *Testing Deployments*—Simplifies testing multiple versions of the same container image.

While there are other reasons, these are the most common considerations for using a specific container image tag, especially in production. Implementing explicit container tags is outside the scope of this book, but I do recommend it for production deployments.

For development and learning purposes, I think using `:latest` is often a great option because it allows you to quickly update your running containers without having to change the manifest or container-building GitHub Actions workflow. This is why we run `kubectl rollout restart deployment <deployment-name>` to force a rollout with the same manifest. The `kubectl rollout restart` is also a practical command if you ever need to “reset” a Deployment after you make updates related to configuration (e.g., Secrets or ConfigMaps changes do not automatically trigger Deployment rollouts).

While you can manually set a container’s image with the command `kubectl set image deployment/<deployment-name> <container-name>=<image-name>:<image-tag>`, I do not recommend doing so in a GitHub Actions workflow because if your manifest had `:latest` and you used `kubectl set` to use the `:v1`, your manifest would be out of sync. The main reason is that the repo’s manifest yaml file would not match the actual Deployment’s image value, which might cause an unintended rollout or rollouts when running `kubectl apply -f apps/` or something similar.

With this in mind, let’s create our workflow within the `roadtok8s-kube` directory at `./.github/workflows/k8s-apps-deploy.yaml`.

Listing 8.28 GitHub Actions workflow for deploying our apps

```
name: Update K8s App Deployments
on:
  workflow_dispatch:

jobs:
  verify_service_account:
    name: Verify K8s Service Account
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repo Code
        uses: actions/checkout@v4
      - uses: azure/setup-kubectl@v3
      - name: Create/Verify `.kube` directory
        run: mkdir -p ~/.kube/
      - name: Create kubectl config
```



```

run: |
  cat << EOF >> ~/.kube/config
  ${ secrets.KUBECONFIG_SA }
  EOF
- name: Echo pods
  run: |
    kubectl get pods
- name: Echo deployments
  run: |
    kubectl get deployments
- name: Apply App Changes
  run: |
    kubectl apply -f apps/
- name: Rollout Apps
  run: |
    kubectl rollout restart deployment -n default py-deploy
    kubectl rollout restart deployment -n default js-deploy

```

Be sure to commit this workflow along with the rest of your Kubernetes manifests. Once you have it committed, add your kubeconfig file to your repository's Secrets with the name `KUBECONFIG_SA` and the value being the contents of your kubeconfig file. This will allow GitHub Actions to use your kubeconfig file to connect to your Kubernetes cluster.

I'll leave it to you to test and run this workflow. If the workflow fails, be sure to try each of these commands locally to ensure they are working there.

Congratulations! You have now successfully deployed a number of applications to a Kubernetes cluster. While this is a great accomplishment, it's meant to be a launching point into the wider world of Kubernetes beyond the scope of this book. I hope you continue your learning path and build, deploy, and release more applications into the wild.

In the next and final chapter, we will look at two alternatives to Kubernetes for deploying containers: *Docker Swarm* and *HashiCorp Nomad*. While I believe Kubernetes is the best option, it's good to know what else is out there and how it compares to Kubernetes.

Summary

- Kubeconfig files are used to connect to a Kubernetes Cluster with the Kubernetes CLI, `kubectl`.
- The `KUBECONFIG` environment variable can be used to set the location of the kubeconfig file for use with `kubectl`.
- Service Accounts contain permission to access various resources within a cluster. The default service account is unlimited in what it can do and access in the Kubeconfig file.
- Pods are the smallest Kubernetes resource and can be defined and provisioned for fast and efficient container deployment.
- Deployments are used to manage and scale stateless Pods across a cluster of Kubernetes nodes.

- StatefulSets are used to manage a scale stateful Pods and include built-in support for scaling and attaching volumes with scaled Pods.
- Deployments and StatefulSets are designed to ensure a specific number of Pods are running based on the replica count, and if a Pod is deleted, failing, or otherwise not running, a new Pod will be provisioned to meet the replica count.
- Updating or changing a container image within a Deployment or StatefulSet will trigger a Pod rollout, which is a no-downtime replacement of Pods.
- ConfigMaps are reusable resources that store static key-value stores that can be attached to other resources like Pods, Deployments, and StatefulSets as environment variables values and read-only volumes.
- Secrets are exactly like ConfigMaps but obscure the value data from the key-value pairs using Base64 encoding. Secrets are designed to be readable when attached to Pods, Deployments, and StatefulSets as environment variable values and read-only volumes.
- Using third-party services such as HashiCorp Vault, AWS Secrets Manager, or Google Secrets Manager is a good alternative to Kubernetes Secrets to obstruct values beyond Base64 encoding.
- Selectors can be used to filter Pods and other resources by key-value pairs.
- Namespaces can be used to group resources together and can be used to filter resources.
- Services manage internal and external communication to Pods, Deployments, and StatefulSets.
- ClusterIP-type Service resources only allow internal traffic and are not accessible from outside the cluster.
- NodePort Service resources expose a port on each node to route external traffic to Pods.
- The LoadBalancer Service resource uses a cloud-managed load balancer service to route external traffic to Kubernetes nodes and Pods.
- The Service types of ClusterIP, NodePort, and LoadBalancer can direct and load balance internal traffic through an internal DNS endpoint or dynamically provisioned IP address called a Cluster IP.
- Container-to-container communication is possible through the use of Service resources.
- A records, or address records, are used to configure a custom domain to a static IP address.
- GitHub Actions can automate and provision Kubernetes resources with manifest files by using a valid kubeconfig file with a service account with the correct permissions and `kubectl`.

Alternative orchestration tools

This chapter covers

- Running multiple containers with Docker and Docker Swarm
- Modifying Docker Compose files to run on Docker Swarm
- Running multiple containers with HashiCorp Nomad
- Using Nomad's built-in user interface to manage container jobs

A key goal of this book is to help you adopt using containers as your go-to-deployment strategy with any given application. We have seen that Kubernetes is an effective way to run and manage containers in production. While I tend to think that Kubernetes is valuable for projects of all sizes, it can be useful to understand a few outstanding alternatives: Docker Swarm and HashiCorp Nomad.

Docker Swarm is essentially Docker Compose distributed across a cluster of virtual machines. We'll see how the technical implementation works, but it is essentially a slightly modified Docker Compose file that connects to worker machines running Docker. Docker Swarm is a good option for those who are already familiar with Docker Compose and want to scale their applications across multiple virtual machines.

After we look at Docker Swarm, we'll implement a container orchestration tool called HashiCorp Nomad. Nomad uses HashiCorp's proprietary configuration language, called HCL, instead of YAML. This difference in configuration syntax might be a deal breaker for some, but HCL is used across many of HashiCorp's tools, such as Terraform and Vault, so it might be worth learning. Nomad itself works similarly to Docker Swarm, but it has a few key differences that we will see in this chapter.

Performing GitOps (e.g., CI/CD workflows with GitHub Actions) with these tools is outside the scope of this book. However, doing so is not too dissimilar to what we have done throughout this book. Let's get started by creating a cluster of virtual machines and using Docker Swarm.

9.1 **Container orchestration with Docker Swarm**

Traditionally, Docker Compose runs on a single host, such as a cloud-based virtual machine or your local computer. Each service declared in a Docker Compose configuration is typically tied to a single running instance of a container image (e.g., `codingforentrepreneurs/rk8s-py`).

Docker Swarm uses almost the same configuration as Docker Compose (it's actually a legacy Docker Compose configuration) to run containers across multiple virtual machines. Docker Swarm is not designed for local development but rather for production deployments. Luckily, you can use Docker Swarm and Docker Compose interchangeably, as we'll see, which is important for verifying that your configuration works before deploying it to a Docker Swarm cluster.

The terminology of Docker Swarm is as follows:

- *Manager node*—The virtual machine or machines that manage the worker nodes. The manager node essentially tells the worker nodes what to run. By default, a manager node is also a worker node.
- *Worker node*—The virtual machine or machines that run the tasks (e.g., containers) that the manager node tells it to run. A worker node cannot run without a manager node.
- *Task*—This is Docker Swarm's term for a running container. The manager node tells the worker node which task (container) to run. Tasks do not move nodes, so once they are defined, they stay put.
- *Service*—Much like with Docker Compose, a service is the definition of a task to execute on the manager or worker nodes.
- *Stack*—A stack is a collection of services deployed to the cluster. A stack is defined in a Docker Compose file.

To relate these terms to Kubernetes, we can think of them as the following:

- Docker Swarm Manager is like the Kubernetes Control Plane. Just as we used the Linode Kubernetes Engine (as the K8s Control Plane) to manage Kubernetes tasks like scheduling Pods and configuring nodes, the Docker Swarm Manager

performs similar roles, such as job scheduling and cluster management, within the Docker Swarm environment.

- Docker Swarm Worker is like the Kubernetes node. The Docker Swarm Worker runs the tasks (or containers) the Docker Swarm Manager tells it to run. The Kubernetes node runs the Pods (or containers) that the Kubernetes Control Plane tells it to run.
- Docker Swarm Task is like a Kubernetes Pod. The Docker Swarm Task is a running container. The Kubernetes Pod is a running container.
- Docker Swarm service is like a Kubernetes Deployment. The Docker Swarm service is a collection of tasks (or containers) deployed to the cluster. The Kubernetes Deployment is a collection of Pods (or containers) deployed to the cluster.
- Docker Swarm Stack is like a Kubernetes Namespace. The Docker Swarm Stack is a collection of services (or containers) deployed to the cluster. The Kubernetes Namespace is a collection of Pods (or containers) deployed to the cluster.

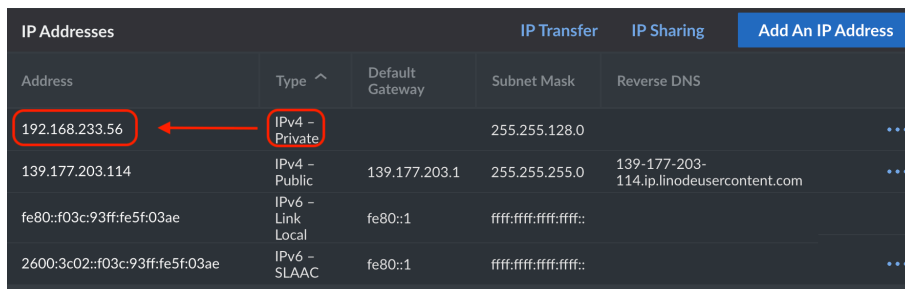
Now that we have some definitions in place, let's provision a virtual machine for a manager node. These steps should look familiar to you, as we have done this before with one exception (private IP addresses):

- 1 Create an account on Akamai Connected Cloud at linode.com/rk8s (book readers receive a promotional credit as well).
- 2 Login to Akamai Connected Cloud (formally Linode; <https://linode.com/>).
- 3 Click the drop-down menu and select Create > Linode.
- 4 Use the following settings:
 - *Distribution*—Ubuntu 22.04 LTS (or latest LTS)
 - *Region*—Seattle, WA (or the region nearest to you)
 - *Plan*—Shared CPU > Linode 2 GB (or larger)
 - *Label*—rk8s-swarm-manager-server (important label)
 - *Tags*—<optional>
 - *Root password*—Set a good one
 - *SSH keys*—<recommended>. Review appendix D to learn how to create one and add it to Linode.
 - *Attach a VLAN*—<skip>
 - *Add-ons*:
 - Backups—<optional>
 - Private IP—Selected
- 5 Click Create Linode.

If you forget to select Private IP, you can add it later by doing the following:

- 1 Navigate to your instance detail page.
- 2 Click the Network tab.
- 3 Click Add an IP Address.
- 4 Under IPv4, select Private IP.
- 5 Click Allocate (If you cannot click this button, you may already have a private IP address attached to your instance).

Before we SSH into this virtual machine, let's make note of the private IP address that has been provisioned. In the detail page of your instance, navigate to the Network tab and scroll to the bottom of the page. Figure 9.1 shows the private IP address for our instance.



IP Addresses		IP Transfer		IP Sharing	Add An IP Address
Address	Type ^	Default Gateway	Subnet Mask	Reverse DNS	
192.168.233.56	IPv4 - Private		255.255.128.0		
139.177.203.114	IPv4 - Public	139.177.203.1	255.255.255.0	139-177-203-114.ip.linodeusercontent.com	
fe80::f03c:93ff:fe5f:03ae	IPv6 - Link Local	fe80::1	ffff:ffff:ffff:ffff::		
2600:3c02::f03c:93ff:fe5f:03ae	IPv6 - SLAAC	fe80::1	ffff:ffff:ffff:ffff::		

Figure 9.1 Private IP address for our virtual machine instance

Now we have three key attributes about our virtual machine:

- The public IP address (for our SSH session)
- The private IP address (for our Docker Swarm server node)
- The Docker Swarm Manager server node label (e.g., rk8s-swarm-manager-server)

With these attributes in mind, let's prepare our virtual machine as a Docker Swarm manager node.

9.1.1 *Preparing the Docker Swarm Manager*

For our Swarm cluster, we will have three total virtual machines: one manager and two worker nodes. After configuring the worker nodes, we'll use the *manager* as our primary agent (or node) to manage the cluster via a swarm-specific `compose.yaml` file. In other words, we only make changes to the swarm via the manager node; the worker nodes just do what they are told.

SSH into our newly created virtual machine using the public IP address and install Docker; listing 9.1 shows these steps succinctly. Be sure to replace `<public-ip-address>` with the public IP address of your virtual machine.

Listing 9.1 Installing Docker engine on Ubuntu

```
ssh root@<public-ip-address> | Shows the standard SSH command
curl -fsSL https://get.docker.com | sudo bash | This is a one-line shortcut to
                                                    install Docker via an official script.
```

Once these items are complete, verify Docker is working by running `docker ps`, and you should see that no containers are currently active. If this portion fails, please revisit previous chapters to review.

With Docker installed, we are going to configure three services: an NGINX load balancer, our containerized Python app (`codingforentrepreneurs/rk8s-py`), and our containerized Node.js app (`codingforentrepreneurs/rk8s-js`). Each of these services will correspond to the following names:

- `web`—Our Python app
- `api`—Our Node.js app
- `nginx`—Our NGINX load balancer

In Docker Compose, we will declare these names as the service names, as we have before. Docker Compose's built-in DNS resolver allows service discovery via the service name as the hostname. Put another way, the service `web` can be accessed via `http://web:8080` (assuming port 8080 is used) from within any given Docker Compose-managed container.

Since we can use service names as DNS host names, we will configure NGINX to distribute traffic to each service and load balance traffic as designed. If you use different service names, be sure that both the NGINX configuration and the Docker Compose configuration match.

From within your SSH session and directly on the virtual machine, create a file named `nginx.conf` with the contents of listing 9.2. Normally, I would advise that we use Git and a CI/CD pipeline to deploy this configuration file, but for the sake of simplicity, we're going to do it directly on the virtual machine.

Listing 9.2 NGINX configuration file for load-balancing Docker Compose services

```
upstream web_backend {
    server web:8080; | Web is the Docker Compose
                    | service name for our Python app.
}

server {
    listen 80; | The default NGINX port is 80.

    location / {
        proxy_pass http://web_backend; | web_backend will forward traffic to all
        proxy_set_header Host $host;    | servers defined in the upstream block
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /api {
```

```

proxy_pass http://api:3000;
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}

```

Api is the Docker Compose service name for our Node.js app.

While this NGINX configuration will be effective, it is one of the simpler configurations by design to ensure our focus remains on Docker Swarm. Docker Swarm also has built-in mechanisms for load balancing that can work hand in hand with the NGINX container image.

Now create the file `compose.yaml` on your server with Docker Compose installed. The following listing shows us exactly that with a configuration we saw earlier in this book.

Listing 9.3 Docker Compose file before Docker Swarm

```

services:
  nginx:
    image: nginx
    ports:
      - 80:80
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf
    depends_on:
      - web
      - api
  web:
    image: codingforentrepreneurs/rk8s-py
    environment:
      - PORT=8080
  api:
    image: codingforentrepreneurs/rk8s-js

```

Verify this configuration by running `docker compose up` or `docker compose -f compose.yaml up` (for a more verbose version). Once each service has started, visit your server's IP address on your local machine's web browser (e.g., `http://<your-ip-address>`). Visiting the root path (`/`) should display the containerized Python application (e.g., `codingforentrepreneurs/rk8s-py`). Visiting the `/api` path should display the containerized Node.js application (e.g., `codingforentrepreneurs/rk8s-js`).

As a reminder, NGINX is a powerful tool that has a lot of configuration options. In our simple example, NGINX forwards requests and the requested path to the appropriate service. This means that the `/api` path in NGINX forwards to the `/api` path in our Node.js application. There are ways to modify this behavior through the NGINX configuration, but doing so is outside the scope of this book.

Assuming that our Docker Compose configuration is working correctly, let's take it down either with `Control + C` or the command `docker compose down`. The reason we verify Docker Compose works first is because Docker Swarm is basically a super-charged Docker Compose for a cluster of virtual machines.

Docker Swarm uses a legacy version of Docker Compose's configuration file, so we'll need to make a few modifications to our `compose.yaml` file to make it work with Docker Swarm.

9.1.2 Docker Compose for Docker Swarm

The Docker Compose file that we need for Docker Swarm requires a few extra layers of configuration.

The first one is how we bind local files to a service. In our case, how do we bind the `nginx.conf` file to the `nginx: service`? We must modify `volumes: from - ./nginx.conf:/etc/nginx/conf.d/default.conf` to what is in listing 9.4.

Listing 9.4 Volume definition for Docker Swarm

```
version: '3.8' # legacy Docker Compose version
services:
  nginx:
    ...
    volumes:
      - type: bind
        source: ./nginx.conf
        target: /etc/nginx/conf.d/default.conf
```

This modification may work with Docker Compose because it's a legacy syntax for Compose file version 3. Review the Docker Compose documentation (<https://docs.docker.com/compose/compose-file/compose-file-v3/>) for more information on version 3's syntax.

Since Docker Swarm is a container orchestration tool, we can define the number of replicas (or instances) of each service we want to run in the `deploy:` directive. The more instances you run, the more compute power your cluster will need, which is exactly why we can continue to add node(s) to our swarm.

On your Docker Swarm Manager instance, create the file `compose-swarm.yaml`, which includes the `deploy:` directive blocks and replica counts you want for each service.

Listing 9.5 Deploy blocks for Docker Swarm

```
version: '3.8' # legacy Docker Compose version
services:
  nginx:
    image: nginx
    ports:
      - 80:80
    volumes:
      - type: bind
        source: ./nginx.conf
        target: /etc/nginx/conf.d/default.conf
    deploy:
      replicas: 1
  web:
    image: codingforentrepreneurs/rk8s-py
```

```
environment:
  - PORT=8080
deploy:
  replicas: 5
api:
  image: codingforentrepreneurs/cfe-nginx
  deploy:
    replicas: 2
```

As you can see, this swarm-ready legacy Docker Compose file is not too different from what we have seen before. In fact, you could run `docker compose -f compose-swarm.yaml up`, and it would work just fine, although the replicas in the `deploy:` directive would simply be ignored.

Now that we have our Docker Compose file ready for Docker Swarm, let's start the Swarm cluster.

9.1.3 **Start Docker Swarm**

Let's not forget: Docker Swarm is for running multiple containers across multiple virtual machines. At this point, we only have one virtual machine we designated as the Docker Swarm Manager.

To start a swarm, we need our private IP address so that other instances can join the swarm and communicate with this virtual machine. We could use a public IP address, but that may be a security risk. It will also cost more since the communication would leave the network go through the internet to then come back to the network.

Here's how we initialize the swarm:

- 1 SSH into the Docker Swarm Manager instance.
- 2 Run `docker swarm init --advertise-addr <private-ip-address>`.

That's it. After you run this command, you should see a message like figure 9.2 shows. This output indicates how other nodes can join the swarm with `docker swarm join --token <some-token> <private-ip-address>:2377`. You can also get this token by running `docker swarm join-token worker` on the Docker Swarm Manager instance.

```
root@localhost:~# docker swarm init --advertise-addr 192.168.216.115
Swarm initialized: current node (9hbuymz7lvr9yg9508f0jxjpe) is now a
manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-2xg2lpl05p3qjgtzs71qi05jucp8
3msmmvmordwixpx745w-cw1hgh0lzyjt1on1roupstxx4 192.168.216.115:2377
```

To add a manager to this swarm, run `'docker swarm join-token manager'` and follow the instructions.

Figure 9.2 Docker Swarm initialization output

Now that we have a swarm initialized, let's add a few nodes to the cluster. To do this, we need to do the following:

- 1 Create two new virtual machines in the same region (required for private IP communication) with private IP addresses and labeled `rk8s-swarm-worker-1` and `rk8s-swarm-worker-2`.
- 2 SSH into each of the new virtual machines.
- 3 Install Docker Engine on each of the new virtual machines with `curl -fsSL https://get.docker.com | sudo bash`.
- 4 Use `docker swarm join --token <some-token> <private-ip-address>:2377` to join the swarm. Running this command will result in a statement like: *This node joined a swarm as a worker*. Repeat this step for each of the new virtual machines.

The number of worker nodes is up to you. I think a good starting point for your cluster is three nodes (including the manager). The previous steps are the same for each node you want to add to the cluster.

Now that we have our manager and worker nodes set up and our swarm-ready compose file, it's time to deploy our Docker Swarm Stack.

9.1.4 Deploy a Docker Swarm Stack

A Docker Swarm Stack is a lot like running `docker compose up` but for a cluster of machines. The options we have are as follows:

- `docker stack deploy -c <compose-file> <stack-name>`—This command will deploy a stack to the swarm. The `-c` flag is the compose file to use, and the `<stack-name>` is the name of the stack. The stack name is used to identify the stack in the swarm; you must pick a name yourself
- `docker stack ls`—This command will list all the stacks you have running in the swarm. Yes, you can have many of them.
- `docker stack ps <stack-name>`—This command will list all the tasks (or containers) running in the stack. This is a good way to see if your stack is running correctly.
- `docker stack rm <stack-name>`—This command will remove the stack from the swarm. This will not remove the images from the nodes, just the stack.
- `docker ps`: This command will *still* list all the containers running on the node. This command works for both the manager and worker nodes.
- `docker exec <container-id> /bin/bash`—This is how you can enter the Bash shell of any given running container. This command is *the same* as we saw before. The key reason this is listed is to mention that you must run this command on the node in which the task (or container) is running.

Now, in your SSH session on the Docker Swarm Manager instance, run `docker stack ls`, and you should see no stacks running. Now run `docker stack deploy -c compose-swarm.yaml rk8s`, and the output should be as you see in figure 9.3.

```
root@localhost:~# docker stack deploy -c compose-swarm.yaml rk8s
Creating network rk8s_default
Creating service rk8s_nginx
Creating service rk8s_web
Creating service rk8s_api
```

Figure 9.3 Docker Swarm Stack deployment output

With this stack deployed, you can now run `docker stack ls` and see the `rk8s` stack listed with three services. You can also run `docker stack ps rk8s` to see the tasks (or containers) running in the stack.

If you visit the public IP address of *any* of the virtual machines in your Docker Swarm cluster, you should see the same results as you did with just Docker Compose, only this time you have many more instances of each service running.

When you need to make changes to this Docker Swarm Stack, you can adjust the `compose-swarm.yaml` file and run `docker stack deploy -c compose-swarm.yaml rk8s` again. Docker Swarm will only update the services that have changed.

Docker Swarm is a great way to quickly spin up containers across many different virtual machines. Docker Swarm does build on the knowledge of Docker Compose, so it's a good idea to be familiar with Docker Compose before using Docker Swarm.

The big advantage Docker Swarm has over Kubernetes is how simple it is to move from Docker Compose to Docker Swarm since they use a lot of the exact same configuration. This configuration is also simple and does not require a steep learning curve. That said, Kubernetes, and especially managed Kubernetes, offers the huge benefit of quickly provision static IP addresses, load balancers, persistent storage volumes, and other cloud resources. Docker Swarm does not have this capability built-in. Kubernetes also has a massive ecosystem of tooling that is Kubernetes-specific and not available to Docker Swarm.

The challenge I will leave you with is how to automate configuring new instances in your swarm. Here's a hint: it has something to do with GitHub Actions. Now let's look at another approach of orchestrating containers with HashiCorp Nomad.

9.2 *HashiCorp Nomad*

Nomad is a rather large departure from the various tools we have used throughout this book. While Nomad is designed to run containers, the configuration is based on HashiCorp's own configuration language (HCL) rather than YAML. HCL is closer to JSON than YAML, but it is still a declarative language used in many popular tools like HashiCorp Terraform and HashiCorp Vault.

For some, using HCL is a deal-breaker for adopting Nomad. For others, it will fit nicely with their existing HashiCorp projects (e.g., Terraform, Vault, Consul).

For our purposes, we'll look at a basic example of using Nomad after we configure a cluster of virtual machines.

9.2.1 Preparing our Nomad cluster

Before we create our cluster, let's look at a few key terms related to Nomad:

- *Server*—The server is the primary agent that manages the cluster. The server is responsible for maintaining the state of the cluster and scheduling tasks (or containers) to run on the client nodes.
- *Client*—The client is the virtual machine that runs the tasks (or containers) the server tells it to run. A client cannot run without a server.
- *Task*—This is Nomad's term for a running container. The server tells the client which task (container) to run. Once defined, a task remains on the same node and does not move node to node.
- *Job*—A job is a collection of tasks deployed to the cluster. A job is defined in a Nomad configuration file.

To relate these terms to Kubernetes, we can think of them as the following:

- Nomad Server is like the Kubernetes Control Plane. Just as we used the Linode Kubernetes Engine (as the K8s Control Plane) to manage Kubernetes tasks like scheduling Pods and configuring nodes, the Nomad Server performs similar roles, such as job scheduling and cluster management, within the Nomad environment.
- Nomad Client is like the Kubernetes Node. The Nomad Client runs the tasks (or containers) that the Nomad Server tells it to run. The Kubernetes Node runs the Pods (or containers) that the Kubernetes Control Plane tells it to run.
- Nomad Task is like a Kubernetes Pod. The Nomad Task is a running container. The Kubernetes Pod is a running container.
- Nomad job is like a Kubernetes Deployment. The Nomad job is a collection of tasks (or containers) deployed to the cluster. The Kubernetes Deployment is a collection of Pods (or containers) deployed to the cluster.

To prepare our Nomad cluster, we need to create three virtual machines: one to act as the Nomad server and two to act as Nomad clients. Here are the steps you should take to do so:

- 1 Create an account on Akamai Connected Cloud at linode.com/rk8s (book readers receive a promotional credit as well).
- 2 Log into Akamai Connected Cloud (formally Linode; <https://linode.com/>).
- 3 Click the drop-down menu and select Create > Linode.
- 4 Use the following settings:

- *Distribution*—Ubuntu 22.04 LTS (or latest LTS)
 - *Region*—Seattle, WA (or the region nearest to you)
 - *Plan*—Shared CPU > Linode 2 GB (or larger)
 - *Label*—rk8s-nomad-1 (increment as needed)
 - *Tags*—<optional>
 - *Root password*—Set a good one
 - *SSH keys*—<recommended>. Review appendix D to learn how to create one and add it to Linode.
 - *Attach a VLAN*—<skip>
 - *Add-ons*:
 - Backups—<optional>
 - Private IP—Selected
- 5 Click Create Linode.

After a few minutes, you should have three virtual machines with the following:

- Instances labeled as rk8s-nomad-1, rk8s-nomad-2, and rk8s-nomad-3
- A public and a private IP address per instance

9.2.2 *Installing Nomad*

Unlike Docker, HashiCorp has no official shortcut script to install Nomad on a virtual machine. Because of this, I created an open source version of this book based on the official installation steps in the HashiCorp Nomad documentation (<https://developer.hashicorp.com/nomad/docs/install>). To review the script, you can visit the Road to Kubernetes Nomad repo (<https://github.com/jmitchel3/roadtok8s-nomad>).

At a minimum, we need to install both Nomad and Docker Engine on each of our virtual machines. To do this, perform the following steps:

- 1 SSH into the virtual machine.
- 2 Install nomad by running


```
curl -fsSL https://raw.githubusercontent.com/jmitchel3/roadtok8s-nomad/main/install-nomad.sh | sudo bash
```
- 3 Install Docker Engine with `curl -fsSL https://get.docker.com | sudo bash`.
- 4 Verify Nomad is installed with `nomad version` and Docker Engine is installed with `docker version`.

Repeat this process for each of your virtual machines. Once you have Nomad and Docker Engine installed on each of your virtual machines, you're ready to configure Nomad.

9.2.3 Configuring the Nomad server

The Nomad server is the so-called brains of the Nomad cluster. For more advanced configurations, we can distribute these brains across multiple virtual machines, including across regions, but for our purposes, we'll just use one virtual machine as the Nomad server.

The default configuration of Nomad, assuming it's installed correctly, should be available to you at the location `/etc/nomad.d/nomad.hcl`. This default config is more of a placeholder for the various configurations you may need. We won't go into the specifics of all these options; instead, we'll cover the critical few to ensure our cluster is set up. Before you replace yours, you might consider reviewing it to see what the default configuration looks like and what options are available to you.

Before we see the completed configuration, here are the declarations we will make and what they mean:

- `data_dir = "/opt/nomad/data"`—This is the default location for Nomad to store data about the cluster.
- `bind_addr = "0.0.0.0"`—Using `0.0.0.0` on any machine means that it will listen on all available IP addresses (This is true for *any* process bound to this same IP address). This value ensures Nomad is running on *any public or private IP address*.
- `name = "nomad-server-1"`—You can pick the name of your nomad server. I recommend incrementing the name as you add more servers.
- `advertise` (our private IP address)—This is the IP address that other nodes in the cluster will use to communicate with this node; we will use three sub-attributes (`http`, `rpc`, `serf`) to ensure our Nomad clients can communicate with this Nomad server.
- `server` with `enabled = true` and `bootstrap_expect = 1`—These settings will ensure this Nomad instance will run as a server and that it should expect only one Nomad server.
- `client` with `enabled = false`;—While you can run a Nomad Server as a client, it's not recommended.

With this in mind, let's start creating our first HCL configuration file for Nomad. Here are the steps to do so:

- 1 SSH into the Nomad server instance.
- 2 Create the directory `/opt/nomad/data` with `mkdir -p /opt/nomad/data`.
- 3 Remove the default configuration with `rm /etc/nomad.d/nomad.hcl`.
- 4 Create the file `/etc/nomad.d/nomad.hcl` with the contents of listing 9.6. Be sure to replace the listed private IP address of `192.168.204.96` with your Nomad server's private IP address.

Listing 9.6 Nomad server configuration

```
# Located at /etc/nomad.d/nomad.hcl
data_dir = "/opt/nomad/data"

bind_addr = "0.0.0.0"

name = "nomad-server-1"

advertise {
  http = "192.168.204.96"
  rpc = "192.168.204.96"
  serf = "192.168.204.96"
}

server {
  enabled = true
  bootstrap_expect = 1 # number of servers
}

client {
  enabled = false
}
```

If you are familiar with JSON (JavaScript Object Notation), you can see some of the inspiration for HCL. HCL uses curly braces (`{}`) to denote a block of configuration, square brackets (`[]`) to denote a list of values, and the `=` sign to denote a key-value pair. YAML, on the other hand, uses indentation to denote a block of configuration, a dash (`-`) to denote a list of values and a colon (`:`) to denote a key-value pair.

With this file created, we can run Nomad in two different ways:

- As a background service with `systemctl` (recommended)
- As a foreground service with `nomad agent`

When comparing the Nomad CLI to the Kubernetes CLI (`kubectl`), it's important to note that the Nomad CLI is not nearly as full-featured `kubectl`. The Nomad CLI focuses on job scheduling, task management, and basic monitoring. `kubectl`, on the other hand, has a variety of complex operations that can happen as well as a variety of plugins that can be used to extend its functionality.

Running Nomad as a foreground service is a great way to test your configuration, but it's not recommended for production. Running Nomad as a background service is the recommended way to run Nomad in production. Luckily for us, the configuration for the background service is already included:

- 1 Enable the background service with `sudo systemctl enable nomad`. Enabling is required one time.
- 2 Start it with `sudo systemctl start nomad`.
- 3 If you make changes to Nomad, run `sudo systemctl restart nomad`. (If the changes do not propagate, run `sudo systemctl daemon-reload` before restarting.)

- 4 Verify the background service with `sudo systemctl status nomad`.
- 5 A way to help diagnose problems with the background Nomad service is to run `sudo journalctl -u nomad`.

With Nomad running in the background, run `nomad server members`, and you should see output similar to figure 9.4.

```
root@localhost:~# nomad server members
Name          Address      Port  Status  Leader  Raft  Version  Build  Datacenter  Region
nomad-server-1.global  192.168.204.96 4648  alive   true    3      1.6.1    dc1      global
```

Figure 9.4 Nomad server members output

The key values to look for are the following:

- **Name**—This should be the name you declared in the configuration with `.global` appended to it.
- **Address**—This needs to be your Nomad server’s private IP address (likely not `192.168.204.96` as shown).
- **Status**—This value should be `alive`; otherwise, we have a problem with our configuration.
- **Leader**—This value will be set to `true` since it is the only Nomad server we have. If we had more than one, each server member would automatically elect a leader, and this value would be `false` for all but one server member.
- **Datacenter**—We can modify this value, but `dc1` is the default value and what you should see here.

Before we configure our Nomad Clients, let’s access the Nomad server UI to verify it is also working. To do so, we need the Nomad server’s Public IP Address and visit port `4646` because this is the default port. Open your browser to `http://<public-ip-address>:4646/`, and you will be redirected to `/ui/jobs` and see the Nomad UI, as shown in figure 9.5.

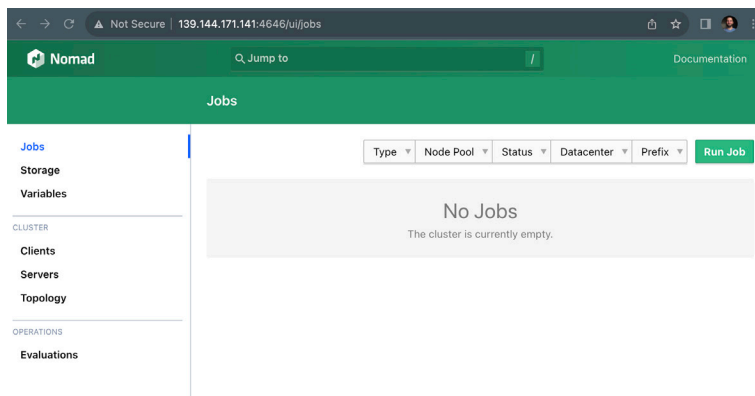


Figure 9.5 Nomad server user interface

The Nomad user interface is a great way to manage our Nomad cluster. Soon we will use the UI to create our first job and run our first containers with Nomad. We could also use the command line, and in some scenarios, we definitely would, because the Nomad UI is publicly accessible by default. For the purposes of this simple example, using the UI is sufficient. Before creating jobs, we need to configure our Nomad Clients. Let's do that now.

9.2.4 **Configuring the Nomad Clients**

Nomad Clients have the same installation process as the Nomad Server (e.g., both Nomad and Docker installed) but the `nomad.hcl` configuration is slightly different. The following listing shows the configuration file for our first Nomad Client, which we'll call `nomad-client-a`.

Listing 9.7 Nomad Client configuration

```
# Located at /etc/nomad.d/nomad.hcl

data_dir = "/opt/nomad/data"

bind_addr = "0.0.0.0"

name = "nomad-client"

server {
  enabled = false
}

client {
  enabled = true
  options = {
    "driver.raw_exec.enable" = "1"
    "docker.privileged.enabled" = "true"
  }
  servers = ["192.168.204.96:4647"]
  server_join {
    retry_join = ["192.168.204.96:4647"]
  }
}

ui {
  enabled = false
}
```

We have two new declarations in this configuration file:

- `client` with the subattributes
 - `enabled = true`: This is required for the client to run.
 - `options = { }`
 - `driver.raw_exec.enable = true`: This is required for the client to run.
 - `docker.privileged.enabled = true`: This is required for the client to run.

- `servers = ["<nomad-server-private-ip>:4647"]`: This is the private IP address of the Nomad server. This is how the client will communicate with the server.
- `server_join`:
 - `retry_join = ["<nomad-server-private-ip>:4647"]`: Once again, this is the private IP address of the Nomad server. This will ensure the client continues to try to join the server until it is successful.
- `server` with `enabled` being `false` thus not attempting to run this instance as a server.
- `ui` with `enabled` being `false`, thus not exposing the UI on this instance. Use this on any Nomad Server or Client you want to disable the UI.

With these concepts in place, let's configure our Nomad Client. Here are the steps to do so:

- 1 SSH into your Nomad Client instance.
- 2 Install Nomad with


```
curl -fsSL https://raw.githubusercontent.com/jmitchel3/roadtok8s-nomad/main/install-nomad.sh | sudo bash
```
- 3 Install Docker with `curl -fsSL https://get.docker.com | sudo bash`.
- 4 Create the directory `/opt/nomad/data` with `mkdir -p /opt/nomad/data`.
- 5 Remove the default configuration with `rm /etc/nomad.d/nomad.hcl`.
- 6 Create the file `/etc/nomad.d/nomad.hcl` with the contents of listing 9.7.
 - Change `name="nomad-client"` to `name="nomad-client-a"` (or `nomad-client-b` for the second client).
 - Replace the listed private IP address of `192.168.204.96` with your Nomad server's private IP address.
- 7 Enable and start the background service with `sudo systemctl enable nomad` and `sudo systemctl start nomad`.

After you complete this step with both clients, SSH back into your Nomad server and run `nomad node status`. Figure 9.6 shows the output of this command.

```
root@localhost:~# nomad node status
ID           Node Pool DC   Name           Class Drain Eligibility Status
0155f4da     default  dc1  nomad-client-b <none> false eligible  ready
686a9eb9     default  dc1  nomad-client-a <none> false eligible  ready
```

Figure 9.6 Nomad server node status output

You can also verify this data in the Nomad UI (e.g., `http://<nomad-server-public-ip-address>:4646/ui/`) and navigate to the Clients tab as seen in figure 9.7.

ID	Name	State	Address	Node Pool	Datacenter	Version
0155f4da	nomad-client-b	ready	192.168.136.90:4646	default	dc1	1.6.1
686a9eb9	nomad-client-a	ready	192.168.237.157:4646	default	dc1	1.6.1

Figure 9.7 Nomad server UI clients

Now that we have the Nomad server and the Nomad Clients, it's time to create our first Nomad job.

9.2.5 *Running containers with Nomad jobs*

To run containers with Nomad, we must define a task using HCL. Task definitions live within group definitions, and group definitions live within Nomad job definitions (again, as HCL). Put another way, each unique job can have many groups, and each unique group within a job can have many tasks (or containers). The general layout is shown in the following listing.

Listing 9.8 Nomad job definition

```
job "a_unique_job_name" {
  group "unique_group_per_job" {
    task "unique_task_per_group" {
      driver = "docker"
      config {
        image = "codingforentrepreneurs/rk8s-py"
      }
    }
  }
}
```

The definitions in listing 9.8 are a valid job, but it's missing a few key configurations (more on this soon). Unsurprisingly, we need a Docker runtime (like Docker Engine) to run the container, and we need a valid and publicly accessible container image (e.g., `codingforentrepreneurs/rk8s-py`); both of these items are declared in the task definition. Private container images can run, too, but they require additional configuration that is outside the scope of this book.

To test this job configuration let's create a new job on our Nomad server. Here are the first few steps to do so:

- 1 Open your Nomad server at `http://<nomad-server-public-ip-address>:4646/ui/`.
- 2 Click the tab `Jobs`.
- 3 Click the button `Run Job`.

At this point, you can paste in the code from listing 9.8, as you can see in figure 9.8.

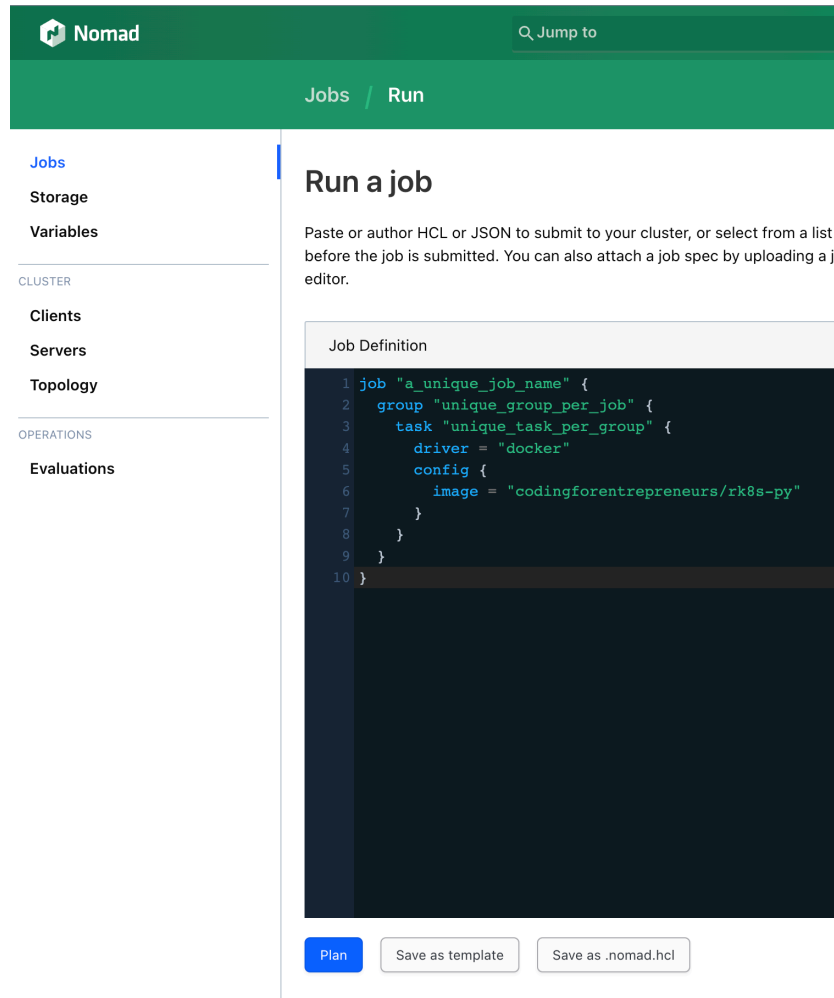


Figure 9.8 Nomad server UI run job

After you click `Plan`, you will see the output of what the job intends to do. HashiCorp tools like Terraform are great at letting you know what will happen before you agree to make it happen. This simple plan is shown in figure 9.9.

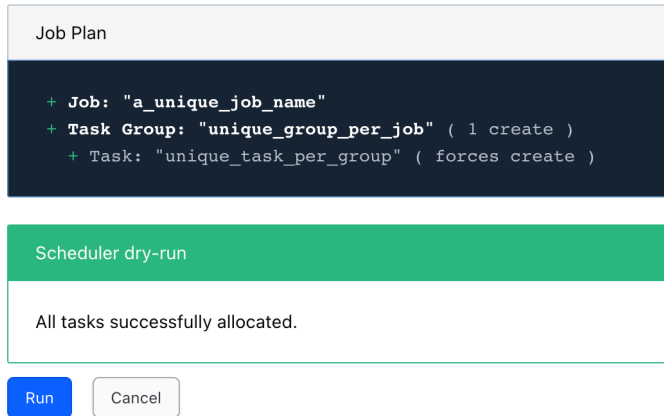


Figure 9.9 Nomad server UI plan job

Now click Run and watch as Nomad kicks off the process of attempting to run this container image. If done correctly, you will see the same output as in figure 9.10.

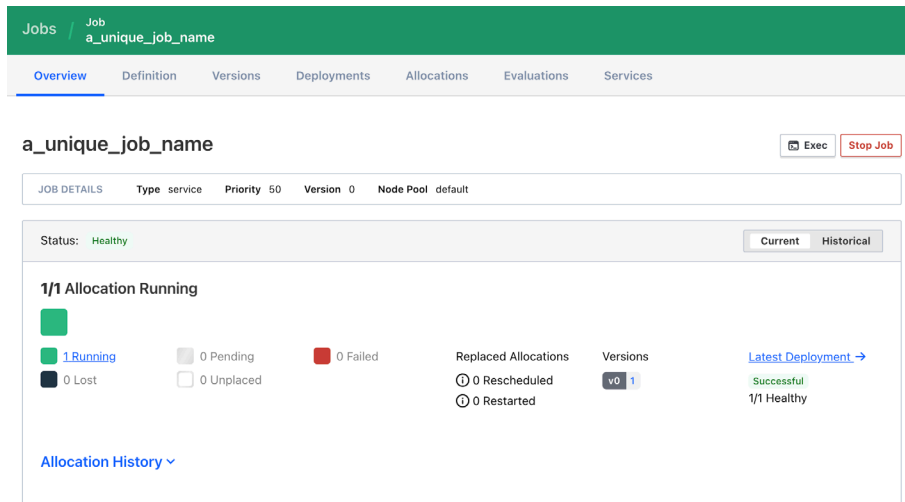


Figure 9.10 Nomad server UI job detail

Learning the ins and outs of the Nomad server UI is out of the scope of this book, but I do encourage you to explore it. The Nomad server UI is a great way to rapidly iterate over jobs and learn more about the Nomad cluster.

Now that we have a job running, let's remove it altogether in the Nomad UI. The steps are as follows:

- 1 Go to your Nomad server UI at `http://<nomad-server-public-ip>:4646/ui/`.
- 2 Click the tab `Jobs`.
- 3 Find the job you want to stop (i.e., `a_unique_job_name`).
- 4 Click `Stop Job` and Click `Yes, Stop Job` to confirm.
- 5 Click `Purge Job` to remove the job from the Nomad server.

If you are interested in the command-line equivalent of this process, it's the following steps:

- 1 SSH into your Nomad server.
- 2 Create a Nomad HCL file, like `nomad-job.hcl`, with the contents of your job definition (see listing 9.8).
- 3 Run `nomad job plan nomad-job.hcl` to see the plan.
- 4 Run `nomad job run nomad-job.hcl` to run the job.
- 5 Run `nomad job status <job-name>` to see the status of the job.
- 6 Run `nomad job stop <job-name>` to stop the job; include the flag `-purge` to completely remove the job from Nomad. If you do not purge the job, you can run `nomad job start <job-name>` to start the job again.
- 7 Repeat this process for each job you want to run or manage.

Creating and removing jobs is simple and a good way to get practice doing so.

Now let's create another job with an additional container and two replicas of each container. The following listing shows the new job definition.

Listing 9.9 Job definition with two tasks for two containers

```
job "web" {
  group "apps" {
    count = 1
    task "unique_task_per_group" {
      driver = "docker"
      config {
        image = "codingforentrepreneurs/rk8s-py"
      }
    }
    task "js-app" {
      driver = "docker"
      config {
        image = "codingforentrepreneurs/rk8s-js"
      }
    }
  }
}
```

As we can see, we have two tasks within one group. This group also has `count = 1`, which is the default value. I put this here so we can practice modifying a pre-existing job. Let's change the value to `count = 2` and see what happens. Here are the steps to do so:

- 1 Open your Nomad server at `http://<nomad-server-public-ip-address>:4646/ui/`.
- 2 Click the tab `Jobs`.
- 3 Select the job you want to modify (i.e., `web`).
- 4 Look for the tab `Definition`.
- 5 After clicking `Definition` you should see the previous job definition.
- 6 Click `Edit`.
- 7 Edit the definition as you see fit (e.g., `count = 2`).
- 8 Click `Plan`.
- 9 Click `Run`.

After a few moments, the modified plan should execute successfully in full. Now our two containers should be running two times each across our Nomad client instances. To verify this, I SSHed into both of my Nomad Clients, ran `docker ps`, and found the output shown in figure 9.11.

```
root@localhost:~# docker ps
CONTAINER ID   IMAGE                                COMMAND                                  CREATED
STATUS        PORTS          NAMES
4e4a464be16d   codingforentrepreneurs/rk8s-js     "docker-entrypoint.s..." 1 se
cond ago      Up Less than a second   js-app-8cad8477-21bd-5e4c-b2cf
-612b2d95ae3e
6a5588f760ce   codingforentrepreneurs/rk8s-py     "./entrypoint.sh"          20 s
econds ago    Up 19 seconds          unique_task_per_group-8cad8477
-21bd-5e4c-b2cf-612b2d95ae3e

root@localhost:~# docker ps
CONTAINER ID   IMAGE                                COMMAND                                  CREATED
STATUS        PORTS          NAMES
369b6765f65f   codingforentrepreneurs/rk8s-py     "./entrypoint.sh"          35 seconds
ago           Up 34 seconds          unique_task_per_group-2e06adfe-2e17-d648-4a44-5827
ebca3e30
0c4b6d844c22   codingforentrepreneurs/rk8s-js     "docker-entrypoint.s..." 38 seconds
ago           Up 37 seconds          js-app-2e06adfe-2e17-d648-4a44-5827ebca3e30
```

Figure 9.11 Nomad Client Docker PS

Adding `count = 2` to our group told Nomad to run each task (container) twice across our cluster so that there are four total container instances running. In this case, Nomad found that the most effective way to do this was to distribute one of each container image to both Nomad Clients. This is not always the case, but it is the case for this example.

An important piece that is missing from our tasks is the PORT mapping we want our containers to have. For this, we can introduce a new block within our group definition, `network`, as shown in the following listing.

Listing 9.10 Nomad job with network port mapping

```
job "web" {
  group "apps" {
    ...
    network {
      port "http-py" {
        to = 8080
      }
      port "http-js" {
        to = 3000
      }
    }
  }
}
```

Using the `network` block within our group gives each task within that group access to those values. Let's see this in action by updating our tasks with the correct port mapping as seen in the following listing.

Listing 9.11 Nomad job definition with two tasks for two containers with network port mapping

```
job "web" {
  group "apps" {
    ...
    task "py-app" {
      ...
      config {
        image = "codingforentrepreneurs/rk8s-py"
        ports = ["http-py"]
      }
    }
    task "js-app" {
      ...
      config {
        image = "codingforentrepreneurs/rk8s-js"
        ports = ["http-js"]
      }
    }
  }
}
```

As usual, mapping the internal container ports correctly is an important step for running them effectively, but this brings up a new question: How can we access these containers? Yes, we could run SSH sessions into each Nomad client to access the containers, but we want to be able to publicly access them.

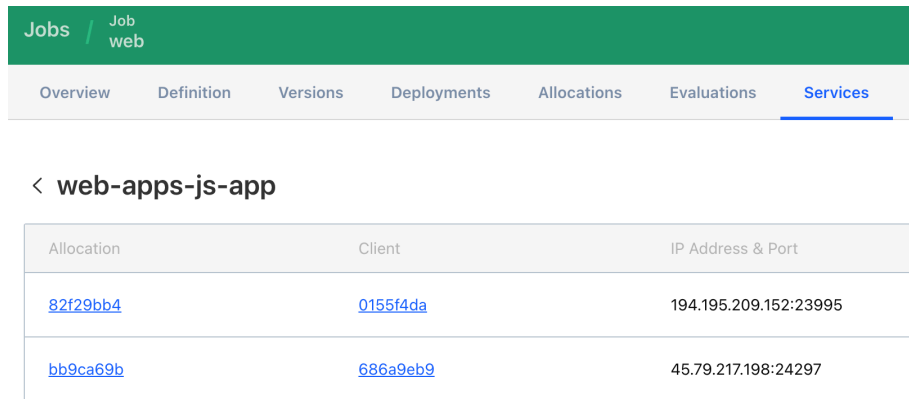
To expose our tasks (containers) to the internet with Nomad and our current configuration, we have two options: a group-level service or a task-level service. For our tasks, we must use the task-level service because of our port values. If our port values were the same for each service, we could then use a group-level service.

A service will automatically assign a port (at random) that is mapped to the container port (e.g., `http-py` and `http-js`) assigned to each running instance of the task. This means that if we have two instances of the `unique_task_per_group` task, we will have two ports assigned to that task. The same is true for the `js-app` task. To create a service, we need to add a new block to our task definition, `service`, as shown in listing 9.12.

Listing 9.12 Nomad tasks with service definitions

```
job "web" {
  group "apps" {
    ...
    task "py-app" {
      ...
      service {
        port = "http-py"
        provider = "nomad"
      }
    }
    task "js-app" {
      ...
      service {
        port = "http-js"
        provider = "nomad"
      }
    }
  }
}
```

Once you update the job definition, navigate to the *Services* tab for this job, and you should see two new services with two allocations. Click on either service, and you will see the exposed port and IP address of each service and task, as seen in figure 9.12.



The screenshot shows the HashiCorp Nomad server UI. At the top, there's a green header with 'Jobs' and 'Job web'. Below it is a navigation bar with tabs: Overview, Definition, Versions, Deployments, Allocations, Evaluations, and Services (which is highlighted). The main content area shows a breadcrumb '< web-apps-js-app'. Below this is a table with three columns: Allocation, Client, and IP Address & Port. The table contains two rows of data.

Allocation	Client	IP Address & Port
82f29bb4	0155f4da	194.195.209.152:23995
bb9ca69b	686a9eb9	45.79.217.198:24297

Figure 9.12 Nomad server UI service details

On this screen, we see the values

- 194.195.209.152:23995
- 45.79.217.198:24297

These are the public IP addresses of our Nomad Clients and the ports Nomad assigned to each task's service. These ports are dynamically allocated to ensure our job definitions do not conflict with each other or with our group counts (e.g., `count = 2`). If you increase the count in a group or add additional groups or additional jobs, Nomad is intelligent enough to allocate port addresses that are available to be used for any given service.

If you use HashiCorp's other tools, like Terraform or Vault, using Nomad is very logical before jumping into using Kubernetes. Nomad is a much simpler tool to use overall and has better cross-region and cross-datacenter support over the default Kubernetes configuration. Nomad's UI is, in my opinion, far easier to use than the Kubernetes Dashboard. Using managed Kubernetes, like mentioned with Docker Swarm, allows you to quickly provision cloud resources like load balancers, persistent storage volumes, and static IP addresses. Nomad does not have this capability built-in, but if you were to couple it with HashiCorp's Terraform tool, it would be more than possible to do so.

At this point, we have covered the basics of Nomad for managing and deploying a number of containers across a number of machines. To unleash the full power of Nomad, we would need to tightly couple it with another service by HashiCorp called Consul. Consul gives Nomad the ability to do service discovery similar to the built-in features of Kubernetes and Docker Swarm. Configuring Consul is outside the scope of this book, but I encourage you to explore it on your own.

Summary

- Docker Swarm is a lot like Docker Compose, but it runs on multiple virtual machines.
- Docker Swarm and Docker Compose are almost interchangeable, but Docker Swarm requires more configuration.
- Docker Swarm has built-in service discovery, making it easier to run multiple containers across multiple virtual machines.
- HashiCorp Nomad has an intuitive user interface that makes managing the cluster, the jobs, and the clients easier.
- HashiCorp has a suite of tools that use HCL, making Nomad a great choice for those already using HashiCorp tools.
- Nomad's dynamic IP allocation allows each running container instance to be publicly accessible in seconds.

appendix A

Installing Python on macOS and Windows

In chapter 2, we create a Python web application. This application will be transformed throughout the book to be deployed to Kubernetes.

Before we can start building our application, we need to install Python on our machine. This appendix will help us do that on macOS and Windows. In chapter 2, we cover how to install Python on Linux. For my most up-to-date guide on installing Python, please consider visiting my blog at <https://cfe.sh/blog/>. If you are interested in the official documentation for installing Python, please visit <https://docs.python.org/>.

A.1 Python installation for macOS

With macOS, you might be tempted to use third-party installation tools like Homebrew or MacPorts. While these tools are great, I recommend using the official Python installer from the Python website simply because it will ensure we have the exact version we need when we need it. We can have multiple versions of Python installed without any issues.

To start, we need to determine two things:

- Python version
- Processor—Intel or Apple

A.2 Python version

Python 3 is the current version of Python. The current major release (as of January 2023) is Python 3.11.

Throughout the years, Python has improved, but the syntax remains largely the same. For this book, I recommend Python 3.10, with a minimum release of Python 3.8.

A.2.1 Processor: Intel or Apple

For newer versions of Python, you can skip to the next section. For older versions of Python, you need to know what kind of processor you are working with. To determine this value, you need to do the following:

- 1 Click the Apple icon in the top left corner.
- 2 Select About This Mac.
- 3 Look for Chip. It will either be Apple or Intel.

If you have an Apple processor (e.g., M1, M2, M3, etc.), you must use the macOS 64-bit universal2 installer selection. If you have an Intel Processor, the macOS 64-bit universal2 installer or the macOS 64-bit Intel-only installer are the selections you can use. Older Python versions may or may not support your current machine.

A.2.2 Download Python

Now that we know which version of Python we want, 3.10, and which processor we have, Intel or Apple, we can finally download the correct installer:

- 1 Visit **Official Python macOS Releases** (<https://www.python.org/downloads/macos/>).
- 2 Under Stable Releases, locate the latest version of Python 3.10 that has *macOS 64-bit universal2 installer* as an option. At the time of this writing, the version is 3.10.9. The full link will resemble <https://www.python.org/ftp/python/3.10.9/python-3.10.9-macos11.pkg>.
- 3 Click the link to download the installer.
- 4 After the download completes, navigate to the Downloads folder on your machine.
- 5 Find the installer and open it. It should be named something like `python-3.10.9-macos11.pkg`.
- 6 Run all the installation options defaults for the installer. Customizing the installation is outside the scope of this book.

With the installer complete, we can verify the installation worked in our command line application called Terminal (located in Applications / Utilities), as shown in the following listing. Open the terminal and type the following commands,

Listing A.1 Terminal application

```
python3.10 -V  
python3 -V
```

Because we *just installed* a new version of Python, `python3` will likely default to that version. If we installed Python 3.9, we would type `python3 -V`, which would return Python 3.9.7.

Since we installed Python 3.10, this command should work. If we installed Python 3.9, we would type `python3.9 -V`.

Building Python from the source

Building Python from the source code is a great way to compile the *exact version* of Python you want on your machine, but it's outside the scope of this book.

The general idea is that you would download the source code from nearly any of the macOS releases (<https://www.python.org/downloads/macos/>) and run various terminal commands to compile and install it on your machine.

The vast majority of readers will skip building from the source because it might over-complicate the installation process. If you're the type of person who builds software from the source code, you probably didn't even open this appendix.

Now that we have Python installed, we can move on to creating a virtual environment in section 3.5.2.

A.3 Python installation for Windows

Installing Python on Windows is best done directly from the Python website (<http://python.org/>). The reason is that the Python website will ensure we have the exact version of Python we need when we need it. We can have multiple versions of Python installed without any issues.

This section will cover a few of the aspects of installing Python on Windows. If you are interested in the official documentation for installing Python, visit <https://docs.python.org/3/using/windows.html>.

A.3.1 Verify System Information

Before we can install Python, we need to verify if our Windows is 64- or 32-bit system. If you have a newer machine, you will likely be using a 64-bit system. Here's how we can check:

- 1 Open the Start menu.
- 2 Type System Information and press Enter.
- 3 Look for the System Type section (see figure A.1).
- 4 If you see x64-based PC, you have a 64-bit system. If you see x86-based PC, you have a 32-bit system.

OS Name	Microsoft Windows 10 Pro
Version	10.0.19045 Build 19045
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	TARS
System Manufacturer	System manufacturer
System Model	System Product Name
System Type	x64-based PC
System SKU	SKU

Figure A.1 Windows system information

A.3.2 Install Python

Now that we know if we have a 64-bit or 32-bit system, we can install Python:

- 1 Go to <https://www.python.org/downloads/windows>.
- 2 Pick Python 3.10.X (replace X with the highest number available).
- 3 For the selected Python version, verify the listing for Windows Installer that matches your System Type (e.g., 64-bit or 32-bit).
- 4 Select to download the version 64-bit or 32-bit to match your system (see figure A.2.).
- 5 Click the link to download the installer.

- [Python 3.10.9 - Dec. 6, 2022](#)

Note that Python 3.10.9 cannot be used on Windows 7 or earlier.

- Download [Windows embeddable package \(32-bit\)](#)
- Download [Windows embeddable package \(64-bit\)](#)
- Download [Windows help file](#)
- Download [Windows installer \(32-bit\)](#)
- **Download Windows installer (64-bit)**

- [Python 3.9.16 - Dec. 6, 2022](#)

Note that Python 3.9.16 cannot be used on Windows 7 or earlier.

- No files for this release.

Figure A.2 Python 64-bit Windows installer on Python 3.10.9

After the installer downloads, open and run it and do the following:

- 1 Select Add Python 3.10 to PATH only if this is the *first version* of Python you have installed.
- 2 Select Customize Installation and then Next.
- 3 Select to Install pip and then Next.
- 4 In Advanced Options, use at least the following configuration (see figure A.3):
 - Install for All Users
 - Add Python to Environment Variables
 - Create Shortcuts for Installed Applications
 - Precompile Standard Library

Customize the install location to be `C:\Python310`, where 310 represents 3.10 as periods (.) cannot be in a folder name in this case. If you're installing Python 3.9, you would use `C:\Python39` instead.

5 Click Install

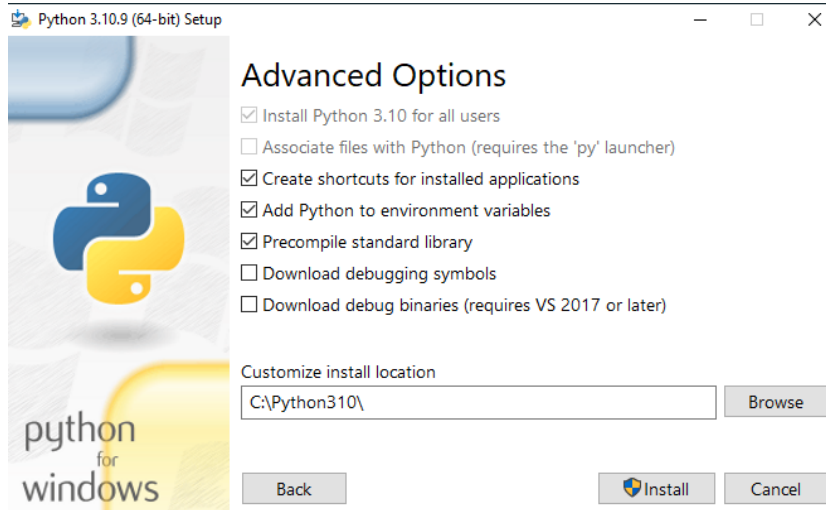


Figure A.3 Advanced Python installation options

Selecting Add Python 3.10 to PATH is also optional but will allow you to shortcut running Python from the command line without specifying the full path to the executable. Open the command prompt and type the following commands in the following listing.

Listing A.2 Using Add Python 3.10 to PATH shortcut

```
python -V
```

← This should yield your most recent Python 3 version installed that ticked the Add Python 3.10 to PATH.

Using the custom installation for `C:\Python310` is optional but recommended because then you can easily have multiple versions of Python on your machine. For example, if you have Python 3.10 and Python 3.9 installed, you can execute either by opening the command prompt and typing the following commands.

Listing A.3 Python 3.10 and Python 3.9 installed

```
C:\Python39\python.exe -V
C:\Python310\python.exe -V
```

Now that you have Python installed on your Windows machine, it's time to create a virtual environment in the section 3.5.2.

A.4 Python installation for Linux

When using Linux, there are many different distributions you can choose from. For installing Python, I am going to assume you're using a Debian-based Linux system—specifically, Ubuntu. Ubuntu has Python 3 installed by default. Open the terminal and type the commands in the following listing.

Listing A.4 Ubuntu with Python 3

```
python3 -V
```

If you see `Python 3.8` or newer, you should be good to skip to the section 3.5.2. If you see `Python 3.8` or older (including Python 3.6 or Python 2.7), you will need to install Python 3.8 or newer. If you need a more in-depth guide on installing Python on Linux, go to chapter 6 because we install Python 3 on Linux for our early production environment for our Python application.

A.5 Virtual environments and Python

Virtual environments are a way to help isolate Python projects from each other and from the system at large. Virtual environments can be easily created, activated, deactivated, and destroyed. Consider virtual environments as ephemeral since we will *never* keep them in our git repos, our Docker images, or our Kubernetes clusters—we always recreate virtual environments when our code is moved around.

To create a virtual environment, we will use a built-in Python module called `venv`. In my opinion, `venv` is the most sustainable way to manage third-party Python packages like FastAPI, Django, requests, and many others. There are many other ways to manage Python packages, but those are outside the scope of this book.

A.5.1 Creating a test project folder

Let's create a single location to hold our test project, as shown in the following listings.

Listing A.5 Creating a project folder on macOS/Linux

```
mkdir -p ~/Dev/k8s-appendix-test
cd ~/dev/k8s-appendix-test
```

Most of my software projects reside in the `Dev` directory in the user's root (`~/`) folder. `k8s-appendix-test` is the name of the directory in which we will create our virtual environment and project. Change them as you see fit.

Navigates to the newly created director

Listing A.6 Creating a project folder on Windows

```
mkdir -p ~\Dev\k8s-appendix-test
cd ~\dev\k8s-appendix-test
```

The command line paths are slightly different for Windows.

Most of my software projects reside in the `~/Dev` directory. `k8s-appendix-test` is the name of the directory in which we will create our virtual environment and project.

A.5.2 Creating a Virtual Environment

Create a virtual environment in the current directory using the recently installed version of Python 3. To do this, we'll use the built-in Python manager called `venv` using the `-m venv <your-environment-name>` flag, as shown in listing A.7. We'll use the name `venv` for our virtual environment. You can use any name you want, but I recommend sticking with one of the following: `venv`, `env`, `_venv`, or `.venv`. If you change it from `venv`, many aspects of this book and the appendixes will not work.

Listing A.7 Creating a virtual environment on macOS/Linux

```
cd ~\Dev\k8s-appendix-test
python3 -m venv venv
```

`python -m` (or `python3 -m`) is how you can call built-in modules to Python; another popular one is using the `pip` module with `python -m pip` (or `python3 -m pip`, depending on your system). This is the preferred way to call `venv` or `pip`, as it will ensure you're using the correct version of Python, as shown in the following listing for Windows.

Listing A.8 Creating a virtual environment on Windows

```
cd ~\Dev\k8s-appendix-test
python -m venv venv
```

In this case, using `python -m` should work if you installed Python 3 correctly on Windows and Python has been added to your PATH (as mentioned previously). In some cases, you may need to use the absolute path to the specific Python version's executable on Windows machines (e.g., `C:\Python310\python.exe -m venv`).

A.5.3 Activating a virtual environment

Activating a virtual environment enables us to use the Python packages we install in that environment. We can activate a virtual environment by opening the terminal and typing the commands in the following listing.

Listing A.9 Activating a virtual environment

```
cd ~/Dev/k8s-appendix-test
source venv/bin/activate
```

The command `source venv/bin/activate` might be new to you. This is how it breaks down:

- `source` is built into macOS and Linux. It's used to execute a script or program.
- `venv` is the name of the directory that holds all of our virtual environment files.
- `bin` is the directory that holds all of the executable files for our virtual environment. You will also find a version of Python here, which you can verify with `venv/bin/python -V`
- `activate` is the executable file that activates our virtual environment.

Open PowerShell and type the commands in the following listing.

Listing A.10 `source venv/bin/activate`

```
cd ~\Dev\k8s-appendix-test
.\venv\Scripts\activate
```

Using `.\venv\Scripts\activate` might be new to you. Here's how it breaks down:

- Run a script on the command line using period (`.`) at the beginning of the command.
- `venv` is the name of the directory that holds all of our virtual environment files.
- `Scripts` is the directory that holds all of the executable files for our virtual environment. You will also find a version of Python here, which you can verify with `venv\Scripts\python -V`.
- `activate` is the executable file that activates our virtual environment.

Activating the virtual environment is likely the primary nuance you'll find when using Python on macOS/Linux and Windows. The rest of the commands are about the same.

A.5.4 *Installing Python packages*

With our virtual environment activated, we can install Python packages using `pip` by opening the terminal and typing the commands as shown in the following listing.

Listing A.11 Using `pip` to install Python packages

```
$(venv) python -V
$(venv) python -m pip install fastapi
$(venv) pip install uvicorn
```

Let's better understand what's happening here:

- `$(venv)` denotes that our virtual environment is active. You might see `#(venv)` or `(venv)` or `(venv) >` depending on your command line application.
- Notice that the `3` is missing from `python`? That's because our activated virtual environment leverages the virtual environment's Python by default. On macOS/Linux, `python3` should still reference the activated virtual environment's Python.
- `python -m pip install` is the preferred way to use `pip` to install third-party Python packages from <http://pypi.org/>.
- `pip install` technically works but is not the preferred way to install third-party Python packages because the command `pip` might be pointing to an incorrect version of Python.

A.5.5 Deactivating and reactivating virtual environments

Whenever you close your command line, your virtual environment will be deactivated by default. If you need to manually deactivate it, you can do so by opening the terminal and typing the commands in the following listing.

Listing A.12 Deactivating a virtual environment

```
$(venv) deactivate
```

← Deactivate is available in an activated virtual environment

Deactivating a virtual environment is often necessary when you need to switch Python projects and/or Python versions.

When you need to reactivate, it's as simple as using one of the commands in the following listings.

Listing A.13 Reactivating a virtual environment, option 1

```
cd ~/Dev/k8s-appendix-test
source venv/bin/activate
```

← Navigates to the directory that holds our virtual environment

← Activates our virtual environment with the original activation command

Listing A.14. Reactivating a virtual environment, option 2

```
cd ~\Dev\k8s-appendix-test
.\venv\Scripts\activate
```

← Navigates to the directory that holds our virtual environment

← Activates our virtual environment with the original activation command

Now let's look at how we can recreate virtual environments.

A.5.6 Recreating virtual environments

At this point, I want to challenge you to recreate your virtual environment from scratch. That is, remove the entirety of the `venv` directory and start over. One of the key things you'll likely need to add is `requirements.txt` to keep track of third-party packages. We use `requirements.txt` in more detail starting in chapter 2.

appendix B

Installing Node.js on macOS and Windows

In chapter 2, we create a Node.js web application. This application will be transformed throughout the book to be deployed to Kubernetes.

Before we can start building our application, we need to install Node.js on our machine. This appendix will help us do that on macOS and Windows. In chapter 3, we cover how to install Node.js on Linux. For my most up-to-date guide on installing Node.js, please consider visiting my blog at <https://cfe.sh/blog/>. If you are interested in the official documentation for installing Node.js, visit <https://nodejs.org/>.

For this book, we just want to make sure we're using the latest long-term support (LTS) version of Node.js. The easiest way to download is by going to <https://nodejs.org> and running through the default installation process. The rest of this appendix will help you install Node.js on macOS and Windows.

B.1 Node.js Installation for macOS

I recommend installing Node.js in one of two ways:

- Directly from Nodejs.org
- Using the Node Version Manager (nvm; <https://github.com/nvm-sh/nvm>)

There are other tools that can install Node.js, but it often makes it more difficult in the long run than the previous two options.

Unlike Python, we do not need to know which processor our Apple computer is running since the latest LTS version of Node.js already supports Apple Silicon (M1, M2, etc.) and Intel processors.

To install nvm, use the following command:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh | bash
```

This command comes directly from the official nvm documentation at <https://github.com/nvm-sh/nvm>.

After nvm is installed, open a new terminal session and run the command `nvm install --lts`. This will install the latest LTS version of Node.js. Once this completes, you should be able to run `node --version` and see the version of Node.js installed.

B.2 Node.js installation for Windows

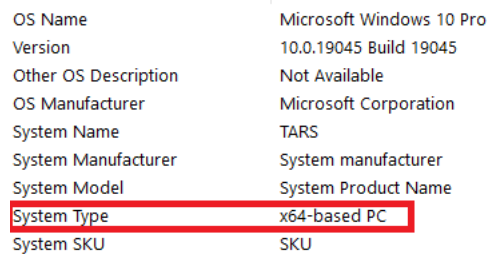
Installing Node.js on Windows is most efficient directly from the Node.js website at <http://nodejs.org>. Node.js has made it simple to download all kinds of versions of nodes, including the latest LTS, which is recommended for most users.

The current LTS is 18.13.0, but the vast majority of Node.js code we'll create in this book should support previous releases of Node.js as well as future releases. Before we can install Node.js, let's verify our system information in the same way we did for Python.

B.2.1 Verifying system information

Before we can install Python, we need to verify if our Windows is 64-bit system or 32-bit. If you have a newer machine, you will likely be using a 64-bit system. Here's how we can check:

- 1 Open the Start menu.
- 2 Type `System Information` and press Enter.
- 3 Look for the `System Type` section (see figure B.1).
- 4 If you see `x64-based PC`, you have a 64-bit system. If you see `x86-based PC`, you have a 32-bit system.



OS Name	Microsoft Windows 10 Pro
Version	10.0.19045 Build 19045
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	TARS
System Manufacturer	System manufacturer
System Model	System Product Name
System Type	x64-based PC
System SKU	SKU

Figure B.1 Windows system information

Now that we know what kind of system we are on, 64-bit or 32-bit, we can install Node.js.

B.2.2 Downloading and installing Node.js

For the vast majority of users, the recommended version of Node.js is the LTS version because it will remain the most well-supported and stable version of Node.js for many years to come. The current LTS version of Node.js is 18.13.0.

If you have a 64-bit machine, you can install Node.js with the following:

- 1 Open a web browser and navigate to <http://nodejs.org/>.
- 2 Click the Download button for the LTS version of Node.js.
- 3 Click to Download the Dfile to Your Computer.
- 4 Double-click the file to start the installation.
- 5 Follow the default prompts to install Node.js.

If you have a 32-bit machine, you can install Node.js with the following:

- 1 Open a web browser and navigate to <http://nodejs.org/>.
- 2 Click the Download button for the LTS version of Node.js.
- 3 Click Other Downloads.
- 4 Under Windows Installer (.msi), click the 32-bit link.
- 5 Click to Download the File to Your Computer.
- 6 After it downloads, double-click to run the installer.
- 7 Follow the default prompts to install Node.js.

If you need to install multiple versions of Node.js on your Windows machine, consider using the tool `nvm-windows` with instructions at <https://github.com/coreybutler/nvm-windows>. Since `nvm` only supports macOS/Linux, `nvm-windows` is a well-made alternative to give you a number of the same features. Installing `nvm-windows` is outside the context of this book.

appendix C

Setting up SSH keys for password-less server entry

In this appendix, we are going to learn how to use a connection protocol known as Secure Shell (SSH) so that our local machine can connect to a remote computer. In this case, a remote computer can be anything from another computer in your house to a virtual machine that you're renting from a cloud provider.

SSH gives us command line access to another computer, which will give us the ability to treat another computer's command line much like we treat our local computer's command line. We very often use SSH to enter Linux-based operating systems, which means your local computer's commands might be slightly different than the Linux-based ones. This is especially true for Windows users because Linux and macOS share a lot of commonality in their operating systems.

To connect to a remote computer with SSH, you often need one of the following:

- A valid username and password
- A public SSH key installed

Given the nature of this appendix, we're going to prepare for using SSH keys as they can allow for password-less entry to a remote computer.

SSH keys are comprised of two components:

- Public key—A shareable key that helps verify our identity
- Private key—A secret, nonsharable key that verifies our public key is, in fact, ours

We can install our public key manually on a large number of servers, we can use `ssh-copy-id` (more on this shortly), or we can have a cloud provider do it for us. In chapter 3, we show an example of how the cloud provider Akamai Connected Cloud (formally Linode) will automatically install our public SSH key for us on any virtual machine we create.

If your SSH key ever fails, you will often have a username and password (just like a lot of software) to log in to the remote device. This username and password can be used in lieu of or in conjunction with valid SSH keys. If you have a username and password but your public SSH key isn't installed, then you can use the command `ssh-copy-id` to install your public SSH key on the remote device. You'll find more on `ssh-copy-id` in the section 3.4.2.

C.1 **Cross-platform SSH key generation**

Since SSH is so widely used, creating SSH keys is commonplace. I encourage you to get used to doing this and, more importantly, get used to deleting these keys and starting from scratch whenever possible.

Here are the primary commands that your command line will likely have:

- `ssh`—Used for all SSH connections.
- `ssh-keygen`—A cross-platform and standardized way to generate a public and private SSH key.
- `ssh-copy-id`—A standardized way to copy public SSH keys to a computer you have private access to.

Let's verify that we have access to using these commands with

```
command -v ssh
command -v ssh-keygen
command -v ssh-copy-id
```

`command -v <some cli command>` is a way to check if a command exists or not.

Most modern command lines have SSH available by default. Some older windows machines require you to download a program called PuTTY to use SSH and related commands.

C.1.1 **Navigating to your root SSH directory**

By default, our SSH keys and configuration will be stored in the `~/.ssh` directory. Let's verify `~/.ssh` exists and navigate to it:

```
mkdir -p ~/.ssh
cd ~/.ssh
```

Working in our user's `.ssh` directory keeps the rest of this guide simple.

C.1.2 **Manual SSH key generation with `ssh-keygen`**

To create an SSH key pair, we use the command `ssh-keygen` with standard options like so:

```
ssh-keygen -t rsa -b 4096
```

You should see something like:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/justin/.ssh/id_rsa):
```

If at any time you need to stop this creation process, just run Control+C. Enter the name `id_rsa_rk8s` and press Enter. You should see:

```
Enter file in which to save the key (/Users/justin/.ssh/id_rsa): id_rsa_rk8s
Enter passphrase (empty for no passphrase):
```

For added security, you can add a passphrase to the SSH key. Think of this like a password for the private SSH key itself.

For simplicity, we're going to leave this blank. Press Enter/Return, and you'll see

```
Enter same passphrase again:
```

Press Enter/Return again, and you'll see

```
Your identification has been saved in /Users/justin/.ssh/id_rsa_rk8s
Your public key has been saved in /Users/justin/.ssh/id_rsa_rk8s.pub
The key fingerprint is:
SHA256:..somekey.. justin@justins-laptop.lan
The key's randomart image is:
+---[RSA 4096]---+
|=.      . = .. +**O.|
|..      = = .. +. |
|O .  B O O      oE |
|O. .+ =          . |
|.. ..=. S         |
|. O O+ O          |
|.. ..O=           |
|.  .+.+O          |
|.  .O=.           |
+----[SHA256]-----+
```

It's important to note that you can run `ssh-keygen` *anywhere* on your computer, and it will create and store the key/pair in the working directory unless you specify an exact path to a storage location. We used `~/.ssh` to help mitigate problems when generating our keys.

C.1.3 Automating ssh-keygen

The following commands will create a new SSH key pair without a passphrase using the file path `~/.ssh/id_rsa_k8s` like we did earlier. The only difference is that these commands do not require any input from the user:

- *macOS/Linux*—`ssh-keygen -t rsa -b 4096 -C "" -f ~/.ssh/id_rsa_k8s -q -N ""`
- *Windows*—`ssh-keygen -b 2048 -t rsa -f ~/.ssh/id_rsa_k8s -q -N ""`

Now that we have created the keys, let's verify they were created in the correct spot.

C.1.4 Verifying the keys

Depending on what file paths you used, your keys might end up in a different location. Let's verify that we stored them in the standard SSH directory for our user:

```
ls ~/.ssh
```

You might see something like this:

```
config
known_hosts
id_rsa
id_rsa.pub
id_rsa_rk8s
id_rsa_rk8s.pub
```

Or, at the very least, you will now see

```
id_rsa_rk8s
id_rsa_rk8s.pub
```

These are the two keys that make up your SSH key pair. The public key We will give our public key (`id_rsa_rk8s.pub`) to our remote hosts, Linode, GitHub, or any other place we might need to use SSH for faster authentication. We need to keep the private key (`id_rsa_rk8s`) safe to prevent unwanted access.

C.2 **Overwriting SSH keys**

The reason we opt to use a new SSH key for this book is to ensure we do not accidentally remove access to other servers and/or expose the wrong keys in the wrong places. What you're learning is that it's incredibly common to delete and/or overwrite your old SSH keys. Let's practice doing so now:

```
cd ~/.ssh
ssh-keygen -t rsa -b 4096
```

You should see something like

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/justin/.ssh/id_rsa):
```

Again, enter the name `id_rsa_rk8s` and press Enter. You should see

```
Enter file in which to save the key (/Users/justin/.ssh/id_rsa): id_rsa_rk8s
id_rsa_rk8s already exists.
Overwrite (y/n)?
```

Feel free to answer `y` or `n` to experiment with what happens. There's no wrong answer here. You can also delete the SSH key pair with

```
rm id_rsa_rk8s
rm id_rsa_rk8s.pub
```

If you start thinking of SSH keys as actually two keys, you're on the right track.

C.3 Copying the public key

Now we'll copy the public SSH key from the `~/.ssh/` directory. It's important to copy the contents of a file that has `.pub` at the end of the file name (e.g., `id_rsa_rk8s.pub`). The public key resembles this string:

```
ssh-rsa <encoded value> justin@justins-laptop.lan
```

Irrelevant data has been omitted, but the entire value (including <encoded value>) should be on one line.

The private key resembles this string:

```
----bash
-----BEGIN OPENSSH PRIVATE KEY-----
b3B1bnNzaC1rZXktdjEAAAABG5vbmUAAAABbm9uZQAAAAAAAAABAAACFwAAAAAdzc2gtcn
...
-----END OPENSSH PRIVATE KEY-----
```

Irrelevant data has been omitted. This file has about 50 lines.

A shortcut to copy the public key is to run the command in the following listings, depending on your system.

Listing C.1 Copying the SSH public key on macOS/Linux

```
cat ~/.ssh/id_rsa_rk8s.pub | pbcopy
```

Listing C.2 Copying the SSH public key on Windows

```
cat ~\.ssh\id_rsa_rk8s.pub | clip
```

Now that we can copy the *public key*, we can share it wherever needed or install it directly.

C.4 Installing the SSH public key on a remote host

For this section, we need an active remote host (such as a Linode virtual machine) we can copy our SSH keys to. We are going to install these keys using two different methods.

C.4.1 Manually installing SSH keys with copy and paste

If your machine has the command `ssh-copy-id`, you might consider skipping to the next step:

- 1 Copy the public key (like listing C.1 or C.2).
- 2 SSH into your remote host: `ssh root@<your_linode_ip_address>`. This should prompt you for a password.
- 3 In the remote host, navigate to `~/.ssh/`: `cd ~/.ssh`
- 4 Create a new file called `authorized_keys`: `touch authorized_keys`
- 5 Open the file: `nano authorized_keys`
- 6 Paste the contents of your public key into the file.
- 7 Close and save the file with `CTRL + X` and then `Y`, and then press `Enter`.

- 8 Review changes with `cat ~/.ssh/authorized_keys`
- 9 Exit the SSH session: `exit`

C.4.2 **Automatically Installing SSH keys with the `ssh-copy-id` command**

The command is simple:

```
ssh-copy-id -i ~/.ssh/id_rsa_rk8s.pub root@<your_server_ip_address>
```

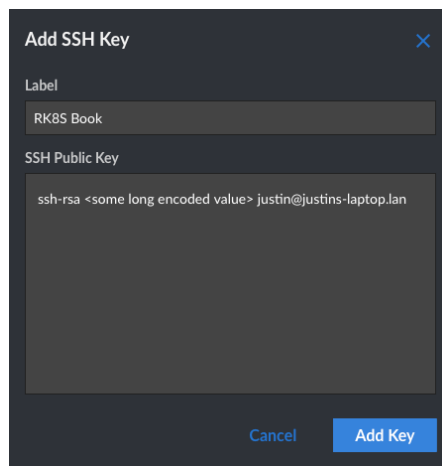
You should see

```
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/Users/
justin/.ssh/id_rsa_rk8s.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to
filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are
prompted now it is to install the new keys
root@<your_server_ip_address>'s password:
```

Once you enter the password, the key should be installed.

C.4.3 **Automatically installing to virtual machines on Linode**

- 1 Go to SSH Keys in the Linode Console (<https://cloud.linode.com/profile/keys>).
- 2 Click the Add an SSH Key button.
- 3 In the Label field, enter `RK8S Book`.
- 4 In the SSH Public Key field, paste your `id_rsa_rk8s.pub` key from the previous section (see figure C.1). The actual key will be longer than what is displayed here.
- 5 Click Add Key.
- 6 Verify the `RK8S Book` key has been added (figure C.2).



The image shows a dark-themed modal window titled "Add SSH Key" with a close button (X) in the top right corner. Inside the modal, there are two input fields. The first field is labeled "Label" and contains the text "RK8S Book". The second field is labeled "SSH Public Key" and contains the text "ssh-rsa <some long encoded value> justin@justins-laptop.lan". At the bottom of the modal, there are two buttons: "Cancel" and "Add Key".

Figure C.1 SSH key example in the Linode console

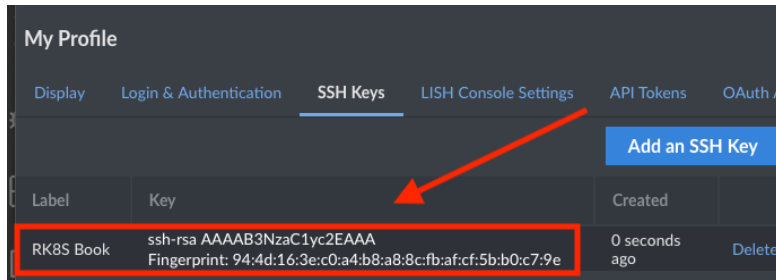


Figure C.2 SSH keys store in the Linode console

Just like when creating these keys on our local machine, you can add or remove them from Linode at will. You can also add multiple keys to your Linode account. This is a great way to ensure you have access to your Linode account from multiple machines.

When you need to install this key on any given virtual machine, just be sure to select to add the `RK`.

C.5 Connect to remote host with a specific SSH key

Regardless of how you installed your SSH public key using the previous methods (sections 3.4.1, 3.4.2, and 3.4.3), we should now be able to use password-less entry to our virtual machine:

```
ssh -i ~/.ssh/id_rsa_rk8s root@<your_linode_ip_address>
```

A few things to note are as follows:

- If your SSH key had a passphrase, you will be prompted for it.
- We are using `-i ~/.ssh/id_rsa_rk8s` and not `-i ~/.ssh/id_rsa_rk8s.pub` with the `ssh` command. Private keys are used to prove a public key is valid.
- We *must* pass `-i ~/.ssh/id_rsa_rk8s` until we update SSH config in the next section.

C.6 Updating or creating SSH config

Writing our private key every time we need to connect to the same server gets old really fast. Sure, if we used the default SSH key, `~/.ssh/id_rsa`, we could omit `-i ~/.ssh/id_rsa_rk8s` altogether, but that might mean we're using the *same public key* everywhere, and that's not great for security.

Instead, let's update our `ssh` config file so we can ensure the specific remote IP address uses the same key every time. Using an admin user, open or edit `~/.ssh/config`:

The diagram illustrates two examples of SSH configuration entries in a `Host` block. Each example is annotated with arrows pointing to specific fields and explanatory text boxes.

Example 1:

```
Host rk8s-book-vm
  Hostname <your_linode_ip_address>
  User root
  IdentityFile ~/.ssh/id_rsa_rk8s
```

Annotations for Example 1:

- An arrow points from the text "Host can be a nickname or a specific IP address." to the `Host rk8s-book-vm` line.
- An arrow points from the text "HostName must be the IP Address or domain name that's mapped to the IP address." to the `Hostname <your_linode_ip_address>` line.
- An arrow points from the text "User must be a user that has SSH access. The root user does by default." to the `User root` line.
- An arrow points from the text "IdentityFile must be a valid path to an SSH private key." to the `IdentityFile ~/.ssh/id_rsa_rk8s` line.

Example 2:

```
Host <your_linode_ip_address>
  Hostname <your_linode_ip_address>
  User root
  IdentityFile ~/.ssh/id_rsa_rk8s
```

Annotations for Example 2:

- An arrow points from the text "Host can be a nickname or a specific IP address." to the `Host <your_linode_ip_address>` line.
- An arrow points from the text "HostName must be the IP Address or domain name that's mapped to the IP address." to the `Hostname <your_linode_ip_address>` line.
- An arrow points from the text "User must be a user that has SSH access. The root user does by default." to the `User root` line.
- An arrow points from the text "IdentityFile must be a valid path to an SSH private key." to the `IdentityFile ~/.ssh/id_rsa_rk8s` line.

Once `~/.ssh/config` is updated, we can now connect to our remote host using the nickname we created or the IP address *without* the `-i` flag for our private key:

```
ssh root@rk8s-book-vm
ssh root@<your_linode_ip_address>
```


appendix D

Installing and using ngrok

We build web applications so that we can share them with others. To share our web applications, we need to make them accessible to the internet.

Going from the applications we have right now into a real production environment will take a few more chapters to fully understand the process, but that doesn't mean we can't put our applications on the internet right now. We can do this by using a tool called `ngrok` (pronounced *en-grok*). `ngrok` is a tool that allows us to expose our local web applications to the internet. It's a great tool for testing and development as well as sharing with friends or coworkers on our progress.

D.1 *Installing ngrok*

We can install `ngrok` with the instructions at their website at <https://ngrok.com/download>. At this location, there is a great step-by-step guide to getting `ngrok` working on your local machine.

D.2 *Verify installation*

Assuming you have installed `ngrok` correctly, you should be able to run the following command:

```
ngrok --version
```

← The output should be `ngrok version 3.1.0.`

D.3 *Running ngrok*

Now that we have verified `ngrok` is installed, let's run it:

```
ngrok http 3011
```

The command breaks down like this:

- `ngrok http`—Start an `ngrok` with an HTTP tunnel.
- `3011`—The port we want to expose, which is exactly mapped to `http://localhost:3011`.

This will output something like the following:

```
stdout
Session Status      online
Account             Justin Mitchel (Plan: Free)
Version             3.1.0
Region              United States (us)
Latency             65ms
Web Interface       http://127.0.0.1:4040
Forwarding           https://1f69-192-63-174-214.ngrok.io -> http://localhost:3011

Connections          ttl      opn      rt1      rt5      p50      p90
                    0         0       0.00    0.00    0.00    0.00
```

Options to Stop/Cancel the running ngrok tunnel are as follows:

- Exit the command line/terminal window (easiest).
- Use CTRL+C in the terminal where the program is running (recommended).
- Use `kill -9 <PID>` where `<PID>` is the process ID of the ngrok tunnel instance (advanced).

A

Akamai Connected Cloud (ACC)

- creating new VM on 81–82

Kubernetes clusters

- provisioning 175

- speed in creating 174

- navigation bar 80

- overview 38

SSH keys

- installing public 79

- pasting new SSH key 80

- purpose-built 78

Andreesen, Marc 3

Ansible

- Ansible Playbook. *See* Ansible Playbook

- automated repeatability 71

- command module 92

- copy, built-in module 95

- current state 85, 90

- declarative programming 85

- deploying Node.js app with 97–102

- deploying Python app with 90–97

- described 71

- desired state 85

- document-based approach to automation 71

- handlers 95

- implementing 96

- reload supervisor 96

- resource for reviewing configuration

- options 85

- running with GitHub Actions 86

- secure shell connection (SSH) 85

- synchronize module 94

- updated state 90

- virtual machine automation with 85–102

Ansible Playbook

- as YAML-formatted file 86

- creating 88–90

- creating for Node.js app 98

- declarative files and 2

- Node.js app 98–101

- playbook_dir variable 94

- Python app 91–97

- vs. GitHub Actions workflow 88

applications

- containerizing 103

- containerized 4, 173

- deploying as complex task 10

- development stage 153

- internet 104

- multiple stages in process of preparing

- applications 153

- portable 104

- production stage 153

- server-side rendered (SSR) 118

- staging 153

- containerized applications 153
- containers and complexity of
 - applications 155
- staging stage 153
- stateful 7
- third-party packages and application
 - development 104
- apt declaration 92
- apt-get install nginx command 82
- APT Package Manager 43
- A record 215, 222
- ARM-based machine 157
- Asynchronous Gateway Interface (ASGI) 57
- automated repeatability 71
- automation, Git repository 47
- AWS Elastic Beanstalk 38
- AWS Secrets Manager, as alternative to Kubernetes
 - Secrets 199

B

- bash shell 200
- buildpacks
 - creating container images 173
 - described 173

C

- <callback> 18
- CI/CD pipelines 126
 - GitHub Actions and 72–85
- cloud computing 38
- clusters
 - defined 178
 - namespaces and managing cluster
 - resources 211
- command line interface (CLI) 10
- commit history 30
- commit messages 30
 - reasons for writing good 30
 - verbose 29
- compose.prod.yaml file 164
- ConfigMaps 222
 - as key-value pairs 178
 - customizing NGINX 193
 - defined 178
 - different use of 196
 - environment variables with 196–199
 - setting environment variables 198

- Consul, HashiCorp service 247
- container builder 108
- container images 104
 - building with Docker Compose 156
 - designing new 108–110
 - .dockerignore 116
 - issue when using Docker Hub for building 127
 - NGINX 105
 - prebuilt, in Docker 105
 - pushing to Docker Hub 121–124
 - running from Dockerfile 110
 - running Python container image 112
- container instance, scaling 167
- container orchestration
 - alternative tools 223–247
 - described 130
 - Docker Compose as first step toward 130
- containers 5–6, 125
 - as lightweight virtual machines 107
 - bash shell 107, 108
 - building and running without Kubernetes 6–7
 - building with GitHub Actions 129
 - container builder 6
 - container image 6
 - container registry 6
 - container runtime 6, 107
 - cross-container communication 141
 - defined 178
 - deletable Pod 208
 - deploying to Kubernetes 185–199
 - described 5, 104
 - entering NGINX container 107
 - ephemeral nature of 135
 - GitHub Actions workflow for building 129–130
 - internal container port 118
 - it, interactive terminal 107
 - Kubernetes and 172
 - limitations of 104
 - providing network for communication
 - between 140
 - scaling with Docker Compose 168
 - staging 153–157
 - stateful 200–214
 - stateless 7, 107–108
 - system-level dependency 104
 - volumes 127
 - volumes and stateful 135–140

- containerized applications, deploying 148–170
- continuous delivery (CD) 102
 - defined 72
 - staging environment 72
- continuous deployment 73
- continuous integration and continuous delivery (CI/CD) 7
- continuous integration (CI) 102
 - automated tests 72
 - defined 72
- control plane 190
- Create Repository button 33
- curl command 150
- current state 85, 90

D

- databases 143
 - containers and 6
- data loss 47
- declarative files
 - in Docker 2
 - infrastructure-as-code tools and 2
 - in Kubernetes 2
- declarative programming 74
- default configuration file 86
- dependencies 4–5, 148
 - application specific 5
 - hardware specific 5
 - installing apps' dependencies 53–62
 - installing NGINX on Ubuntu 44
 - operating system 5
 - third-party 5, 104
 - various 5
 - version specific 5
- Deployments 222
 - as groups of Pods 191
 - container-based, limits of 4
 - container tag and production deployment 219
 - declaring and mounting volumes 194
 - defined 178
 - manual. *See* manual deployment with virtual machines
 - overview 191
 - software 2
- desired state 85, 90
- development environment 153
 - flexibility 154
- dictionary value 13

- Docker
 - ADD command 115
 - as declarative tool 2
 - buildx tool 124
 - built-in process manager 152
 - container images 104, 105
 - COPY command 115
 - declarative files in 2
 - described 6
 - Docker Desktop. *See* Docker Desktop
 - environment variables 117
 - ignoring unwanted files 115
 - installing on Ubuntu 150
 - installing on VM with GitHub Actions 161
 - ports 106
 - third-party GitHub Action tools 127
 - variations of build and push commands 129
- Docker build platform 124
- Docker Compose 130–135
 - automating commands for building and running containers 131
 - building container image with 156
 - built-in apt package manager 151
 - built-in service discovery 227
 - built-in volume management 138
 - compose.yaml, configuration file 130
 - configuration, key options 131
 - databases 143
 - described 127
 - for Docker Swarm 229–230
 - for multiple environments 155
 - for Node.js applications 134–135
 - for production Node.js 165
 - for production Python 165
 - for Python applications 131–134
 - installing on Ubuntu 151
 - installing on VM with GitHub Actions 161
 - limitations of using for production 167
 - missing 134
 - networking 140–147
 - Redis service in 144
 - remote access 167
 - scaling containers 168
 - single host 224
 - tagging images 156
 - two ways to stop 133
 - using dotenv with 146
 - with multiple Dockerfiles 155

- docker compose build command 132
- docker compose down command 132, 141
- Docker Compose services 169
- docker compose up command 132, 141
- Docker Desktop
 - installing 104
 - official installation guide 104
- Docker Engine 173
 - described 150
 - installing on Ubuntu 150
- Docker Hub 105
 - as official Docker registry 122
 - logging in via docker login 162
 - Node.js version tags 118
 - prebuilt and official container images 110
 - Python version tags 110
 - repository page 123
- Docker registry 121
- Docker Swarm 223
 - advantage over Kubernetes 232
 - built-in service discovery 227, 248
 - configuration 224
 - container orchestration with 224–232
 - Docker Compose 229
 - starting 230
 - terminology 224
- Docker Swarm Manager 226–229
- Docker Swarm Stack, deploying 231
- Dockerfile 2
 - building container image from 109
 - creating 109–110
 - designing new container images 108
 - different Dockerfiles for multiple environments 154
 - example of creating configuration file for container 5
 - syntax 109
- DOCKERHUB_REPO 159
- DOCKERHUB_TOKEN 159
- .dockerignore file 115, 119
- dotenv, environment variable file 145, 163

E

- entrypoint script 116–118, 119
- environment
 - different Dockerfiles for multiple environments 154
 - staging environment 72

- environment variables 117, 118, 127
 - abstracted away from application code 145
 - ConfigMap for 197
 - defining 196
 - dotenv files 145
 - PORT value 145
 - with ConfigMaps and Secrets 196–199
- EOF (End Of File) command 45
- Express.js
 - as server-side web framework 16
 - creating application 17–19
 - creating JSON response with 20–21
 - described 15
 - handling routes with 18
 - installing 17
 - requirements for using 16
 - returning JSON data with 20
 - running web server 19
 - used as microservice 20

F

- failed, Pod status 187
- FastAPI 10–15
 - creating FastAPI application 13–14
 - described 10
 - handling multiple routes with 14–15
 - port 8011 14
 - using uvicorn to run 14
- files
 - committing changes 27
 - creating post-receive file 52
- File Transfer Protocol (FTP) 44
- git add 27
- Git status and 24
- ignoring 25
- tracking 26
- untracked 25
- firewalls 68

G

- Git 4
 - as open-source tool 21
 - commit history 30
 - committing changes with git commit command 27
 - downside 30
 - features of 21

- git add and tracking Python files 27
 - overview of fundamental git commands 31–32
 - rebooting Git repos 29
 - sharing codes to and from computers 37
 - status 24
 - tracking code changes 21–32
 - using in projects 23–25
 - verifying that it is installed 22
 - version control 22
 - git checkout command 52
 - git commit command 27–29
 - GitHub
 - configuring local git repo for 34
 - creating account 32
 - creating repository on 33
 - pushing code to 32–35
 - GitHub Actions
 - Actions tab 76
 - building and pushing containers 127–130, 159–160
 - building containers with 129
 - CI/CD pipelines 72–85
 - deploying production containers 157–167
 - declarative programming 74
 - described 73
 - Kubernetes resources 222
 - running Docker Compose in production 162
 - Run workflow drop-down list 76
 - security concern and storing secrets in 83
 - sidebar 76
 - third-party plugins 128
 - GitHub Actions secret
 - adding IP address 82
 - adding Personal Access Token to 128
 - creating 78–79
 - New Repository Secret 79
 - purpose-built SSH keys 78
 - login_action plugin 128
 - GitHub Actions workflow 158
 - Ansible 86–88
 - setup result 88
 - example 73–77
 - flexibility of 75
 - for building containers 129
 - for Node.js and Ansible Playbook 101
 - installing NGINX on remote server with 82
 - items to be defined 74
 - managing configurations 164, 165, 171
 - reviewing the output 77
 - syntax for using stored secrets 83
 - vs. Ansible Playbook 88
 - GitHub Repository 4
 - git init command 23
 - GitLab 23
 - Git repository
 - bare repository
 - creating 48
 - described 47
 - project metadata 51
 - few benefits compared to third-party-services 47
 - initializing 23
 - pushing local code to remote repository 50
 - self-hosted remote 47–53
 - logging in to server with SSH 48
 - git status command 24
 - Google Secret Manager 146
 - as alternative to Kubernetes Secrets 199
 - gunicorn 13
 - baseline configuration for 57
 - described 56
- ## H
-
- HashiCorp 146
 - HashiCorp Nomad 223
 - comparing Nomad CLI to Kubernetes CLI 236
 - configuring
 - HCL, configuration language 224, 232
 - installing Nomad 234
 - intuitive user interface 248
 - Nomad clients 238
 - Nomad job definition 240
 - Nomad server 235
 - overview 232
 - preparing Nomad cluster 233
 - running containers with Nomad jobs 240–247
 - terminology 233
 - testing job configuration, example 240
 - HashiCorp Vault, as alternative to Kubernetes Secrets 199
 - HEAD, creating bare Git repositories 48
 - headers, returning JSON data with Express.js 20
 - health checks 167
 - Heroku 38
 - hooks directory 51

HTML source code 19
 <httpMethod> 18
 hybrid deployments 3

I

images, tagging 153
 imperative programming 74
 Infrastructure as Code (IaC) 40, 174
 ingress management 167
 initContainers, creating StatefulSet and 204
 inventory file 86

J

JavaScript
 as beginner-friendly language 2
 browser-based 15
 server-side 15
 JSON data format 13

K

K8s. *See* Kubernetes
 kubeconfig file 179
 KUBECONFIG variable 183
 setting in VSCode 184
 kubectl, Kubernetes CLI 179
 comparing to Nomad CLI 236
 configuring 183
 control plane 190
 installing 181–183
 kubectl binary on Linux 182
 with Chocolatey on Windows 182
 with Homebrew on macOS 182
 running delete-at-anytime Pod 209
 kubectl rollout command 219
 Kubernetes
 access modes 201
 as declarative tool 2
 connecting to 179–185
 control plane 174
 core concepts and components 177–179
 cross-container communication 208–211
 declarative files in 2
 deploying apps to production 214–221
 LoadBalancer services 215
 steps for 214
 deploying containers to 185–199
 deployment path 3

Deployment resource 191
 described 172
 Docker Compose 127
 environment variables 117
 fundamental challenges with using 7–8
 GitHub Actions and deploying to 219
 K8s Dashboard 179, 180
 manifests
 applying metadata label to resources 188
 described 185
 key features 186
 provisioning Kubernetes cluster 175–176
 reasons for using cloud-based managed
 Kubernetes 174–175
 resource types 186
 Kubernetes Services
 creating first 187–191
 default Kubernetes Service 190
 described 178
 fundamental types 187

L

Linode. *See* Akamai Connected Cloud (ACC)
 Linode Kubernetes Engine (LKE) 176
 load balancing 67, 168, 171
 LoadBalancer services 215–219
 log entry, information in 30
 login_action, GitHub Actions plugin 128

M

manager node 224
 manifests 2. *See Also* Kubernetes, Manifests
 manual deployment with virtual machines 37–70
 message queues, containers and 6
 metadata labels 188
 multi-virtual machine support 167

N

Namespaces
 adding resources to 212
 defined 178
 examples of DNS endpoints for services in
 various 210
 Kubernetes resources 211
 namespaces resource for Redis 212
 not used or declared 212
 New Repository Secret 79

NGINX

- configuration for Python and Ansible 91
- container images 105
- cross-container communication 141
- customizing with ConfigMaps 193
- deploying directly to Ubuntu 151
 - stopping containers and publishing port 153
- different ways to remove 45
- ephemeral VMs 44
- installing on remote server with GitHub Actions workflow 82–85
- installing on Ubuntu 44
- installing on Ubuntu systems 82
- installing with Ansible 89
- load balancing 168
- modifying default HTML page 45
- my-nginx-service 141
- overview 43
- reverse proxy configuration 67
- serving multiple applications 66
- serving static websites 43–47

ngrok tool

- described 269
- installing and using 269–270

node, defined 178**Node.js**

- building containers with Docker Compose 134
- .dockerignore file 119
- installation for macOS 258
- installation for Windows 259
 - downloading and installing Node.js 259
 - verifying system information 259
- server-side JavaScript 15
- starting new project 16
- Supervisor configuration for 64

Node.js applications

- containerizing 118–121
- compose.yaml file 134
- Docker Compose 134
- entrypoint script 119
- stateless 108

Node.js Dockerfile 120**Node Package Manager (npm) 16**

- installing application dependencies, example of 60
- package dependencies 17

node pools 176**nodePort 189****NodePort service 215****Node Version Manager (nvm)**

- installing 59
- installing apps' dependencies 53
- verifying it is installed 60

NVM_VERSION bash variable 59**nvm-windows tool 260****O**

on-premise servers, and software deployed on 3

P**package cash 91****<pathString> 18****pending, Pod status 187****PersistentVolumeClaim (PVC) resource 201****Personal Access Token 128****personal devices, and software deployed on 3****Platform-as-a-Service (PaaS) 38, 173****Pod**

- as smallest configurable unit in Kubernetes 186

- creating new in dashboard 187

- defined 178

- deletable 208

- deletion 203

- deployments 191

- different phases 187

- naming scheme 200, 203

- NGINX-based 186

- ordering 203

- rollout 192

-port flag 14**portability 72****PORT mapping 245****ports 106–107****post-receive Git hook 51**

- as bash script 52

- installing app dependencies and updating 61

- Supervisor 66

- updating for Python app 56

- verifying 52

Procfile, configuration file 173**production**

- container runtime 148

- system-level dependencies 148

- using GitHub Actions to run Docker Compose in 162–167

- production containers
 - building and hosting 158–160
 - built on different machine than production host 157
- programming languages, containers and 6
- public IP address, provisioning 167
- public static IP addresses, managed Kubernetes 174
- Python
 - Ansible, command line tool. *See* Ansible
 - API-first approach 13
 - as beginner-friendly language 2
 - building containers with Docker Compose 131, 133
 - building from source code 251
 - current version 249
 - database containers 143
 - developer ecosystem 10
 - dictionary serialization 13
 - downloading installer 250
 - .gitignore file 25
 - installation for Linux 254
 - installation for macOS 249
 - installation for Windows 251
 - installing all third-party packages 12
 - installing packages 12
 - processor 250
 - project development folder 11
 - pushing container image to Docker Hub 122
 - rebooting Git repos, example of 29
 - reviewing commit history log 30
 - steps for creating new project 11
 - Supervisor configuration for 64
 - third-party installation tools 249
- Python 3, installing 54
 - creating server-side virtual environment 55
 - running Python app 56, 57
 - updating Git hook for Python app 56
- Python applications
 - containerizing 110–118
 - Docker Compose 131
 - entrypoint script 116
 - stateless 108
- Python Dockerfile
 - creating 113–115
 - for browsing 114
 - updating for Redis tools 144
 - version tags 110
- Python interpreters 11

- Python Package Index (PyPI) 110
- Python Package Manager (pip) 12
- Python Package Tool (pip), installing apps' dependencies 53

Q

- Quick Setup section 33

R

- ReadOnlyMany access mode 201
- ReadWriteMany access mode 201
- ReadWriteOnce access mode 201
 - replica limit 203
- Redis
 - as stateful application 212
 - StatefulSet for 213
- REDIS_HOST, environment variable 146
- reload flag 14
- remote repository 31, 34
- remote server, creating and connecting to 38–43
- replica count 192
- replica limit 203
- replicas 204
- repository files 23
- request/response cycle 36, 38
- requirements files 12
 - complex 13
- res.json(), returning JSON data 20
- reverse proxy 66, 67
- role-based access control (RBAC) 7
- running, Pod status 187

S

- scalability 72
- secrets
 - defined 178
 - for environment variables 198
 - for Postgres 204
 - third-party tools and securing secret values 199
- Secure Shell (SSH)
 - authenticity check 41
 - connecting to remote computer with 261
 - connecting via 40
 - example of connection 41
 - described 40
 - Linux-based operating system 261

- navigating to root SSH directory 262
 - requirements for performing SSH 40
 - SSH keys
 - ACC and installing public SSH key 79
 - and GitHub Actions 78
 - components 261
 - connecting to remote host with specific SSH key 267
 - copying public key 265
 - creating new 78
 - cross-platform SSH key generation 262
 - installing SSH public key on remote host 265
 - overwriting 264
 - password-less server entry 261
 - permission denied 88
 - purpose-built 78
 - security concern 78
 - ssh key errors 88
 - ssh-keygen 262
 - verifying 263
 - updating or creating SSH config 267
 - security, self-hosting Git repositories and 47
 - selectors 189, 204, 222
 - selector value 189
 - service accounts 178
 - software
 - deployed 3–4
 - deployment 2, 9
 - cloud computing 38
 - ssh command 83
 - stack, Docker Swarm terminology 224
 - staging environments 154
 - StatefulSets 201
 - defined 178
 - defining key configurations for creating StatefulSet 204
 - for Postgres 205
 - key features 203
 - overview 203
 - static IP addresses 38
 - succeeded, Pod status 187
 - sudo apt install nginx command 82
 - sudo apt update command 82
 - sudo command 43, 150
 - Supervisor
 - background processes 62
 - checking status 65
 - configuration for Python and Ansible 91
 - configuration update 65
 - configuring for apps 63–66
 - creating log directories 64
 - described 62
 - installing 63
 - useful commands 63
 - systemctl 68
- ## T
-
- targetPort 189
 - task, Docker Swarm terminology 224
 - template 204
 - Terraform, declarative files and 2
 - third-party GitHub Action plugins 128
- ## U
-
- Ubuntu
 - Docker Compose 151
 - Docker Engine 150
 - installing Docker 150
 - Uncomplicated Firewall (UFW) 68
 - unknown, Pod status 187
 - unicorn 10, 57
 - described 13
 - running FastAPI applications 14
- ## V
-
- venv
 - as built-in virtual environment manager 10
 - described 10
 - version control 22, 36
 - continuous integration and 72
 - view source, HTML source code 19
 - virtual environment
 - activating 11
 - creating 11
 - creating and activating 255
 - deactivating and reactivating 257
 - described 10
 - Python and 254
 - installing Python packages 256
 - recreating 257
 - server-side, creating 55
 - virtual machines
 - installing Docker with GitHub Actions 161
 - provisioning new 149–150

- virtual machine (VM)
 - Akamai Connected Cloud (ACC) 38
 - creating new VM on 81
 - Ansible and VM automation 85–102
 - ephemeral VMs 44
 - provisioning 39
 - renting 39
 - replacing VM IM address 49
 - virtual services, and software deployed on 3
 - Visual Studio Code (VSCode) 184
 - volumeClaimTemplates, creating new Pods and 204
 - volumes 135–140, 201–214
 - and deployments 201–203
 - block storage volumes 201
 - defined 135, 178
 - described 127
 - detaching 135
 - docker command 127
 - environment variables 127
 - key features 137
 - managed Kubernetes 174
 - managing with Docker Compose 139
 - mixing and matching various volume types 140
 - persistent volumes 201
 - using with Docker 135–137
 - VSCode 25
-
- ## W
-
- Watchtower 4
 - Web Server Gateway Interface (WSGI) 13
 - gunicorn 56
 - Web server tech, containers and 6
 - Windows Subsystem for Linux (WSL) 104
 - worker node 224
-
- ## X
-
- x86-based machine 157
-
- ## Y
-
- YAML format 74