Mastering Docker

Fourth Edition

Enhance your containerization and DevOps skills to deliver production-ready applications

Russ McKendrick

Mastering Docker, *Fourth Edition*

Enhance your containerization and DevOps skills to deliver production-ready applications

Russ McKendrick

Packt

BIRMINGHAM—MUMBAI

Mastering Docker, *Fourth Edition*

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

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

Commissioning Editor: Vijin Boricha
Acquisition Editor: Shrilekha Inani
Senior Editor: Rahul Dsouza
Content Development Editor: Alokita Amanna
Technical Editor: Sarvesh Jaywant
Copy Editor: Safis Editing
Project Coordinator: Neil Dmello
Proofreader: Safis Editing
Indexer: Manju Arasan
Production Designer: Joshua Misquitta

First published: December 2015

Second published: July 2017

Third published: October 2018

Production reference: 1100920

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

ISBN 978-1-83921-657-2

`www.packt.com`

Contributors

About the author

Russ McKendrick is an experienced DevOps practitioner and system administrator with a passion for automation and containers. He has been working in IT and related industries for the better part of 27 years. During his career, he has had varied responsibilities in many different sectors, including first-line, second-line, and senior support in both client-facing and internal teams for small and large organizations. He works almost exclusively with Linux, using open source systems and tools across both dedicated hardware and virtual machines hosted in public and private clouds at N4Stack, which is a Node4 company, where he holds the title of practice manager (SRE and DevOps). He also buys way too many records.

I would like to thank my family and friends for their support and for being so understanding about all of the time I have spent writing in front of the computer. I would also like to thank my colleagues at Node4 and our customers for their kind words of support and encouragement throughout the writing process.

About the reviewers

Paul Adamson has worked as an Ops engineer, a developer, a DevOps engineer, and all variations and mixes of all of these. When not reviewing this book, Paul keeps busy helping companies embrace the AWS infrastructure. His language of choice is PHP for all the good reasons, and even some of the bad, but mainly habit. Paul is CTO for Healthy Performance Ltd, helping to apply cutting-edge technology to a cutting-edge approach to wellbeing, and also runs his own development company, Lemon Squeezee.

Ronald Amosa is a contract cloud, DevOps, and site reliability engineer and has worked for some of the biggest banks and insurance companies in Australia and New Zealand. Born and raised in New Zealand, Ron is of Samoan, Tuvalu, and Chinese heritage. A CompSci dropout from the University of Auckland, Ron has carved out a 17-year IT career doing everything from helpdesk, web development, Linux and systems engineering to cloud infrastructure designing and building CI/CD platforms using K8s. When not behind a computer, Ron trains and competes in Brazilian Jiu-Jitsu, practices and plays the drums, or is out riding his motorcycle.

To my wife, Nikki, thanks for being my best pal and supporting the million and one things I sign myself up for.

To my brothers for always having my back, and most of all to my parents Asora and Henga Amosa, for everything you sacrificed to get me to where I am today. I am forever grateful.

Ronald Amosa

Packt is searching for authors like you

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

Table of Contents

Preface

Section 1: Getting Up and Running with Docker

1

Docker Overview

2

Building Container Images

3

Storing and Distributing Images

4

Managing Containers

5

Docker Compose

6

Docker Machine, Vagrant, and Multipass

Section 2: Clusters and Clouds

7
Moving from Linux to Windows Containers

8
Clustering with Docker Swarm

9

Portainer – A GUI for Docker

10

Running Docker in Public Clouds

11

Docker and Kubernetes

12

Discovering other Kubernetes options

13

Running Kubernetes in Public Clouds

Section 3: Best Practices

14
Docker Security

15
Docker Workflows

16
Next Steps with Docker

Assessments

Other Books You May Enjoy

Preface

Docker is a game-changer when it comes to modern applications' architecture and how they are deployed. It has now grown into a key driver of innovation beyond system administration, and it has had a significant impact on the world of web development and beyond. But how can you make sure you're keeping up with the innovations it's driving? How can you be sure you're using it to its full potential?

This book shows you how; it not only demonstrates how to use Docker more effectively, it also helps you rethink and re-imagine what's possible with Docker.

You will cover basic topics, such as building, managing, and storing images, along with best practices, before delving into Docker security. You'll find everything related to extending and integrating Docker in new and innovative ways. Docker Compose, Docker Swarm, and Kubernetes will help you take control of your containers in an efficient way.

By the end of the book, you will have a broad and detailed understanding of exactly what's possible with Docker and how seamlessly it fits into your local workflow, highly available public cloud platforms, and other tools.

Who this book is for

If you are an IT professional and you recognize Docker's importance in innovation in everything from system administration to web development, but you aren't sure of how to use it to its full potential, this book is for you.

What this book covers

Chapter 1, Docker Overview, discusses where Docker came from and what it means to developers, operators, and enterprises.

Chapter 2, Building Container Images, looks at the various ways in which you can build your own container images.

Chapter 3, Storing and Distributing Images, looks at how we can share and distribute images, now that we know how to build them.

Chapter 4, Managing Containers, takes a deep dive into learning how to manage containers.

Chapter 5, Docker Compose, looks at Docker Compose—a tool that allows us to share applications comprising multiple containers.

Chapter 6, Docker Machine, Vagrant, and Multipass, looks at Docker Machine and other tools that enable you to launch and manage Docker hosts on various platforms.

Chapter 7, Moving from Linux to Windows Containers, explains that, traditionally, containers have been a Linux-based tool. Working with Docker, Microsoft has now introduced Windows containers. In this chapter, we will look at the differences between the two types of containers.

Chapter 8, Clustering with Docker Swarm, discusses how we have been targeting single Docker hosts until this point. Docker Swarm is a clustering technology provided by Docker that allows you to run your containers across multiple hosts.

Chapter 9, Portainer – a GUI for Docker, explains that most of our interaction with Docker has been on the command line. Here, we will take a look at Portainer, a tool that allows you to manage Docker resources from a web interface.

Chapter 10, Running Docker in Public Clouds, is where we look at the various ways in which you can run your containers in public cloud services.

Chapter 11, Docker and Kubernetes, takes a look at Kubernetes. Like with Docker Swarm, you can use Kubernetes to create and manage clusters that run your container-based applications.

Chapter 12, Discovering other Kubernetes options, is where, having used Docker to run Kubernetes locally, we take a look at other options for getting up and running with Kubernetes on your local machine.

Chapter 13, Running Kubernetes in Public Clouds, takes a look at various Kubernetes offerings from the 'big four' cloud providers: Azure, Google Cloud, Amazon Web Services, and DigitalOcean

Chapter 14, Docker Security, takes a look at Docker security. We will cover everything from the Docker host to how you launch your images, where you get them from, and the content of your images.

Chapter 15, Docker Workflows, starts to put all the pieces together so that you can start using Docker in your production environments and feel comfortable doing so.

Chapter 16, Next Steps with Docker, looks not only at how you can contribute to Docker but also at the larger ecosystem that has sprung up to support container-based applications and deployments.

To get the most out of this book

To get the most out of this book, you will need a machine capable of running Docker. This machine should have at least 8 GB RAM and 30 GB HDD free with an Intel i3 processor or above, running one of the following OSes:

- macOS High Sierra or above
- Windows 10 Professional
- Ubuntu 18.04 or above

Also, you will need access to one or all of the following public cloud providers: DigitalOcean, Amazon Web Services, Azure, and Google Cloud.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Docker-Fourth-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Code in Action

Code in Action videos for this book can be viewed at `https://bit.ly/35aQnry`.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/9781839213519_ColorImages.pdf`.

Conventions used

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

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: 'You can view the containers' labels with the following `docker inspect` command.'

A block of code is set as follows:

```
ENTRYPOINT ['nginx']
CMD ['-g', 'daemon off;']
```

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

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
$ curl -sSL https://get.docker.com/ | sh
$ sudo systemctl start docker
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: 'Clicking **Yes** will open the Docker installer, showing the following prompt.'

> **Tips or important notes**
>
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Reviews

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

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

Section 1: Getting Up and Running with Docker

In this section, you will learn how to install, configure, and use Docker to launch both simple and complex containerized applications on a local machine.

This section comprises the following chapters:

Chapter 1, *Docker Overview*

Chapter 2, *Building Container Images*

Chapter 3, Storing and Distributing Images

Chapter 4, Managing Containers

Chapter 5, Launching Multiple Containers Using Docker Compose

Chapter 6, Using Docker Machine,Vagrant, and Multipass

1
Docker Overview

Welcome to *Mastering Docker, Fourth Edition*! This first chapter will cover the **Docker** basics that you should already have a pretty good handle on. But if you don't already have the required knowledge at this point, this chapter will help you get up to speed, so that subsequent chapters don't feel as heavy.

By the end of the book, you will be a Docker master able to implement Docker in your environments, building and supporting applications on top of them.

In this chapter, we're going to review the following:

- Understanding Docker
- The differences between dedicated hosts, virtual machines, and Docker installers/installation
- The Docker command
- The Docker and container ecosystem

Technical requirements

In this chapter, we are going to discuss how to install Docker locally. To do this, you will need a host running one of the three following operating systems:

- macOS High Sierra and above
- Windows 10 Professional

- Ubuntu 18.04 and above
- Check out the following video to see the Code in Action: `https://bit.ly/35fytE3`

Understanding Docker

Before we look at installing Docker, let's begin by getting an understanding of the problems that the Docker technology aims to solve.

Developers

The company behind Docker, also called Docker, has always described the program as fixing the '*it works on my machine*' problem. This problem is best summed up by an image, based on the Disaster Girl meme, which simply had the tagline '*Worked fine in dev, ops problem now*', that started popping up in presentations, forums, and Slack channels a few years ago. While it is funny, it is, unfortunately, an all-too-real problem and one I have personally been on the receiving end of, let's take a look at an example of what is meant by this.

The problem

Even in a world where **DevOps** best practices are followed, it is still all too easy for a developer's working environment to not match the final production environment.

For example, a developer using the **macOS** version of, say, **PHP** will probably not be running the same version as the **Linux server** that hosts the production code. Even if the versions match, you then have to deal with differences in the configuration and overall environment on which the version of PHP is running, such as differences in the way file permissions are handled between different operating system versions, to name just one potential problem.

All of this comes to a head when it is time for a developer to deploy their code to the host, and it doesn't work. So, should the production environment be configured to match the developer's machine, or should developers only do their work in environments that match those used in production?

In an ideal world, everything should be consistent, from the developer's laptop all the way through to your production servers; however, this utopia has traditionally been challenging to achieve. Everyone has their way of working and their own personal preferences—enforcing consistency across multiple platforms is difficult enough when a single engineer is working on the systems, let alone a team of engineers working with a team of potentially hundreds of developers.

The Docker solution

Using Docker for Mac or Docker for Windows, a developer can quickly wrap their code in a container that they have either defined themselves or created as a **Dockerfile** while working alongside a sysadmin or operations team. We will be covering this in *Chapter 2*, *Building Container Images*, as well as **Docker Compose** files, which we will go into more detail about in *Chapter 5*, *Docker Compose*.

Programmers can continue to use their chosen **integrated development environment** (**IDE**) and maintain their workflows when working with the code. As we will see in the upcoming sections of this chapter, installing and using Docker is not difficult; considering how much of a chore it was to maintain consistent environments in the past, even with automation, Docker feels a little too easy – almost like cheating.

Operators

I have been working in operations for more years than I would like to admit, and the following problem has cropped regularly.

The problem

Let's say you are looking after five servers: three load-balanced web servers and two database servers that are in a master or slave configuration dedicated to running Application 1. You are using a tool, such as **Puppet** or **Chef**, to automatically manage the software stack and configuration across your five servers.

Everything is going great until you are told that we need to deploy Application 2 on the same servers that are running Application 1. On the face of it, this is not a problem – you can tweak your Puppet or Chef configuration to add new users, add virtual hosts, pull the latest code down, and so on. However, you notice that Application 2 requires a newer version of the software than the one you are running for Application 1.

To make matters worse, you already know that Application 1 flat out refuses to work with the new software stack and that Application 2 is not backward compatible.

Traditionally, this leaves you with a few choices, all of which just add to the problem in one way or another:

- Ask for more servers? While this tradition is probably the safest technical solution, it does not automatically mean that there will be the budget for additional resources.
- Re-architect the solution? Taking one of the web and database servers out of the load balancer or replication and redeploying them with the software stack for Application 2 may seem like the next easiest option from a technical point of view. However, you are introducing single points of failure for Application 2 and reducing the redundancy for Application 1 as well: there was probably a reason why you were running three web and two database servers in the first place.
- Attempt to install the new software stack side-by-side on your servers? Well, this certainly is possible and may seem like a good short-term plan to get the project out of the door, but it could leave you with a house of cards that could come tumbling down when the first critical security patch is needed for either software stack.

The Docker solution

This is where Docker starts to come into its own. If you have Application 1 running across your three web servers in containers, you may be running more than three containers; in fact, you could already be running six, doubling up on the containers, allowing you to run rolling deployments of your application without reducing the availability of Application 1.

Deploying Application 2 in this environment is as easy as merely launching more containers across your three hosts and then routing to the newly deployed application using your load balancer. As you are just deploying containers, you do not need to worry about the logistics of deploying, configuring, and managing two versions of the same software stack on the same server.

We will work through an example of this exact scenario in *Chapter 5*, *Docker Compose*.

Enterprise

Enterprises suffer from the same problems faced by developers and operators, as they employ both types of profession; however, they have both of these entities on a much larger scale, and there is also a lot more risk involved.

The problem

Because of the risk as well as the fact that any downtime could cost sales or impact reputation, enterprises need to test every deployment before it is released. This means that new features and fixes are stuck in a holding pattern while the following takes place:

- Test environments are spun up and configured.
- Applications are deployed across the newly launched environments.
- Test plans are executed, and the application and configuration are tweaked until the tests pass.
- Requests for change are written, submitted, and discussed to get the updated application deployed to production.

This process can take anywhere from a few days to a few weeks, or even months, depending on the complexity of the application and the risk the change introduces. While the process is required to ensure continuity and availability for the enterprise at a technological level, it does potentially add risk at the business level. What if you have a new feature stuck in this holding pattern and a competitor releases a similar—or worse still—the same functionality, ahead of you?

This scenario could be just as damaging to sales and reputation as the downtime that the process was put in place to protect you against in the first place.

The Docker solution

Docker does not remove the need for a process, such as the one just described, to exist or be followed. However, as we have already touched upon, it does make things a lot easier as you are already working consistently. It means that your developers have been working with the same container configuration that is running in production. This means that it is not much of a step for the methodology to be applied to your testing.

For example, when a developer checks their code that they know works on their local development environment (as that is where they have been doing all of their work), your testing tool can launch the same containers to run your automated tests against. Once the containers have been used, they can be removed to free up resources for the next lot of tests. This means that suddenly, your testing process and procedures are a lot more flexible, and you can continue to reuse the same environment, rather than redeploying or re-imaging servers for the next set of testing.

This streamlining of the process can be taken as far as having your new application containers push through to production.

The quicker this process can be completed, the faster you can confidently launch new features or fixes and keep ahead of the curve.

So, we know what problems Docker was developed to solve. We now need to discuss what exactly Docker is and what it does.

The differences between dedicated hosts, virtual machines, and Docker

Docker is a container management system that helps us efficiently manage **Linux Containers** (**LXC**) more easily and universally. This lets you create images in virtual environments on your laptop and run commands against them. The actions you perform to the containers, running in these environments locally on your machine, will be the same commands or operations that you run against them when they are running in your production environment.

This helps us in that you don't have to do things differently when you go from a development environment, such as the one on your local machine, to a production environment on your server. Now, let's take a look at the differences between Docker containers and typical virtual machine environments:

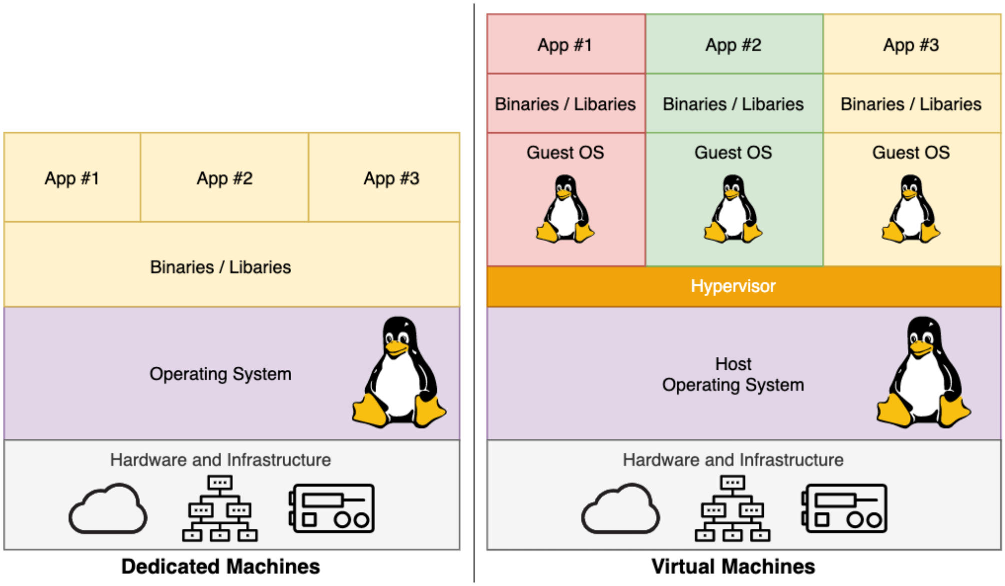


Figure 1.1 – Applications running on virtual machine environments

As you can see, for a dedicated machine, we have three applications, all sharing the same orange software stack. Running virtual machines allows us to run three applications, running two completely different software stacks. The following diagram shows the same three applications running in containers using Docker:

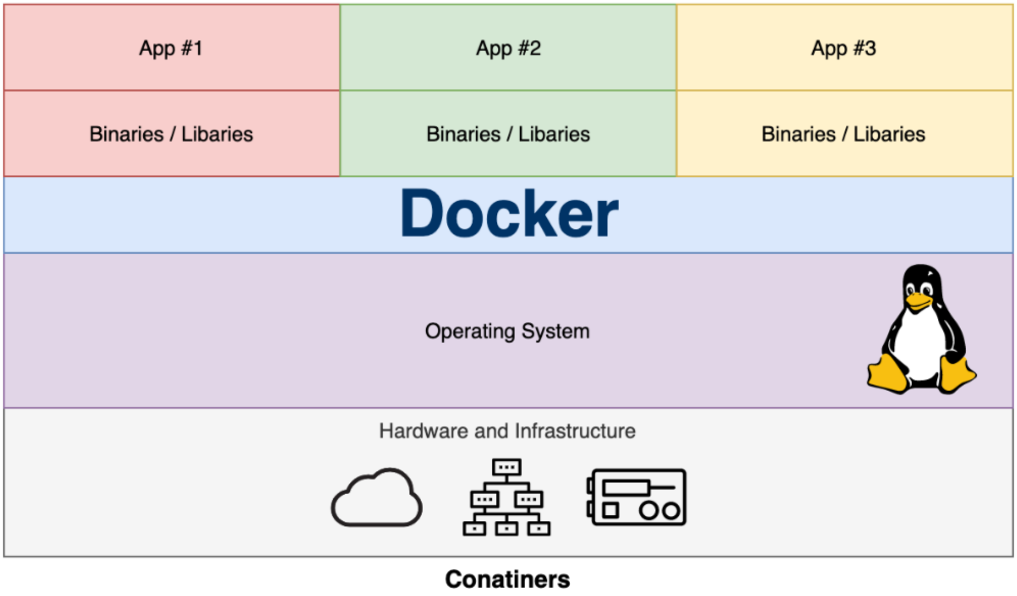


Figure 1.2 – Applications running on top of Docker

This diagram gives us a lot of insight into the most significant key benefit of Docker, that is, there is no need for a complete operating system every time we need to bring up a new container, which cuts down on the overall size of containers. Since almost all the versions of Linux use the standard kernel models, Docker relies on using the host operating system's Linux kernel for the operating system it was built upon, such as Red Hat, CentOS, and Ubuntu.

For this reason, you can have almost any Linux operating system as your host operating system and be able to layer other Linux-based operating systems on top of the host. Well, that is, your applications are led to believe that a full operating system is actually installed—but in reality, we only install the binaries, such as a package manager and, for example, Apache/PHP and the libraries required to get just enough of an operating system for your applications to run.

For example, in the earlier diagram, we could have Red Hat running for the orange application, and Debian running for the green application, but there would never be a need actually to install Red Hat or Debian on the host. Thus, another benefit of Docker is the size of images when they are created. They are built without the most significant piece: the kernel or the operating system. This makes them incredibly small, compact, and easy to ship.

Docker installation

Installers are one of the first pieces of software you need to get up and running with Docker on both your local machine and your server environments. Let's first take a look at which environments you can install Docker in:

- Linux (various Linux flavors)
- macOS
- Windows 10 Professional

Besides, you can run them on public clouds, such as Amazon Web Services, Microsoft Azure, and DigitalOcean, to name a few. With each of these installers listed previously, Docker actually operates in different ways on the operating system. For example, Docker runs natively on Linux. However, if you are using macOS or Windows 10, then it operates a little differently since it relies on using Linux.

Let's look at quickly installing Docker on a Linux desktop running Ubuntu 18.04, and then on macOS and Windows 10.

Installing Docker on Linux

As already mentioned, this is the most straightforward installation out of the three systems we will be looking at. We'll be installing Docker on Ubuntu 18.04; however, there are various flavors of Linux with their own package managers, which will handle this slightly differently. See the *Further reading* section for details on install on other Linux distributions. To install Docker, simply run the following command from a Terminal session:

```
$ curl -sSL https://get.docker.com/ | sh
$ sudo systemctl start docker
```

You will also be asked to add your current user to the Docker group. To do this, run the following command, making sure you replace the username with your own:

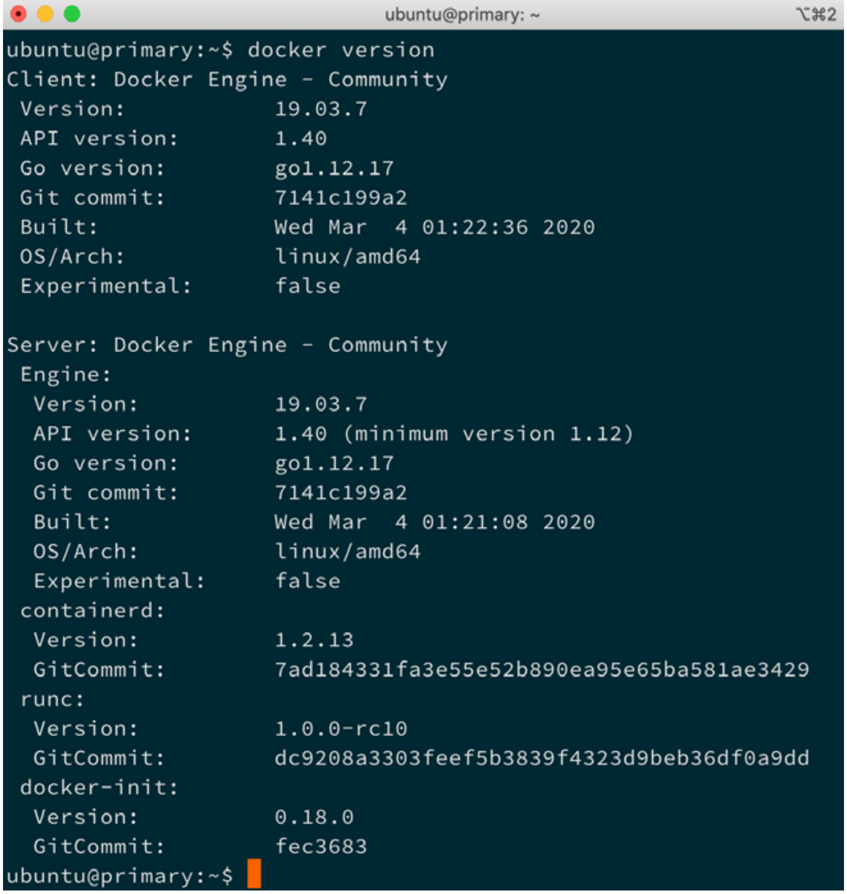
```
$ sudo usermod -aG docker username
```

These commands will download, install, and configure the latest version of Docker from Docker themselves. At the time of writing, the Linux operating system version installed by the official install script is 19.03.

Running the following command should confirm that Docker is installed and running:

```
 $ docker version
```

You should see something similar to the following output:



```
ubuntu@primary:~$ docker version
Client: Docker Engine - Community
 Version:           19.03.7
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        7141c199a2
 Built:             Wed Mar  4 01:22:36 2020
 OS/Arch:           linux/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.7
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.17
  Git commit:       7141c199a2
  Built:            Wed Mar  4 01:21:08 2020
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.2.13
  GitCommit:        7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc:
  Version:          1.0.0-rc10
  GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
ubuntu@primary:~$
```

Figure 1.3 – Output of the docker version command showing the version of Docker installed on the system

There is a supporting tool that we are going to use in future chapters, which are installed as part of the Docker for macOS or Windows 10 installers.

To ensure that we are ready to use the tool in later chapters, we should install it now. The tool is called **Docker Compose**, and to install it we first need to get the latest version number. You can find this by visiting the releases section of the project's GitHub page at https://github.com/docker/compose/releases/. At the time of writing, the version was `1.25.4` – update the version number in the commands in the following code block with whatever the latest version is when you install it:

```
$ COMPOSEVERSION=1.25.4
$ curl -L https://github.com/docker/compose/releases/
download/$COMPOSEVERSION/docker-compose-`uname -s`-`uname -m`
>/tmp/docker-compose
$ chmod +x /tmp/docker-compose
$ sudo mv /tmp/docker-compose /usr/local/bin/docker-compose
```

Once it's installed, you should be able to run the following two commands to confirm the version of the software is correct:

```
$ docker-compose version
```

Now that we know how to install it on Linux, let's look at how we can install it on macOS.

Installing Docker on macOS

Unlike the command-line Linux installation, Docker for Mac has a graphical installer.

> **Tip**
>
> Before downloading, you should make sure that you are running at least Apple macOS X Yosemite 10.10.3 as this is minimum OS requirement to run the version of Docker we will be discussing in this title. If you are running an older version, all is not lost; you can still run Docker. Refer to the *Older operating systems* section of this chapter.

Let's install Docker on macOS:

1. Go to the Docker store at `https://hub.docker.com/editions/community/docker-ce-desktop-mac`.
2. Click on the **Get Docker** link.

3. Once it's downloaded, you should have a `DMG` file. Double-clicking on it will mount the image, and opening the image mounted on your desktop should present you with something like this:

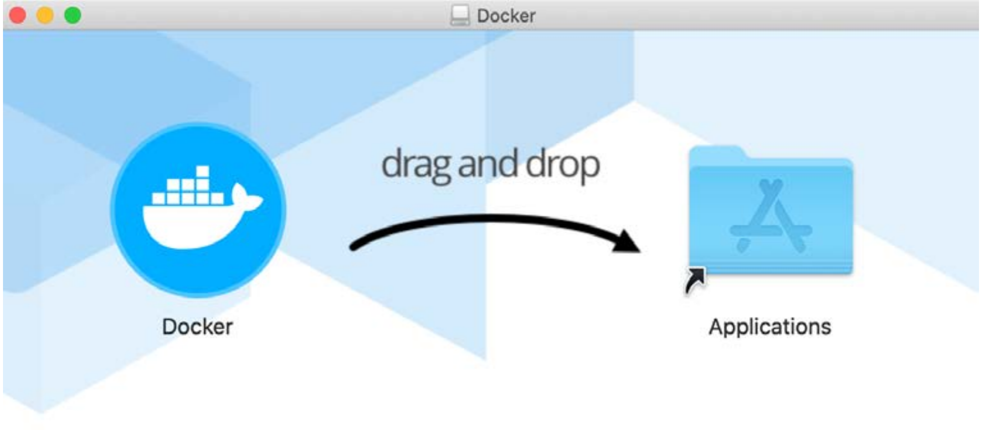
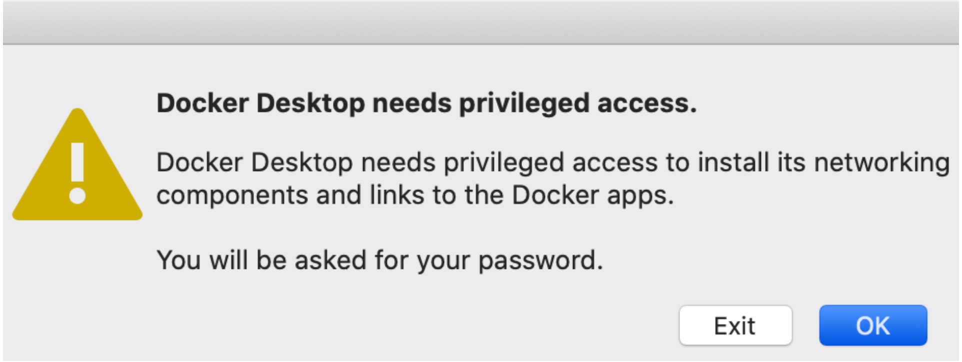


Figure 1.4 – The drag and drop screen for the Docker installer for macOS

4. Once you have dragged the **Docker** icon to your **Applications** folder, double-click on it and you will be asked whether you want to open the application you have downloaded.
5. Clicking **Yes** will open the Docker installer, showing the following prompt:



Docker Desktop needs privileged access.

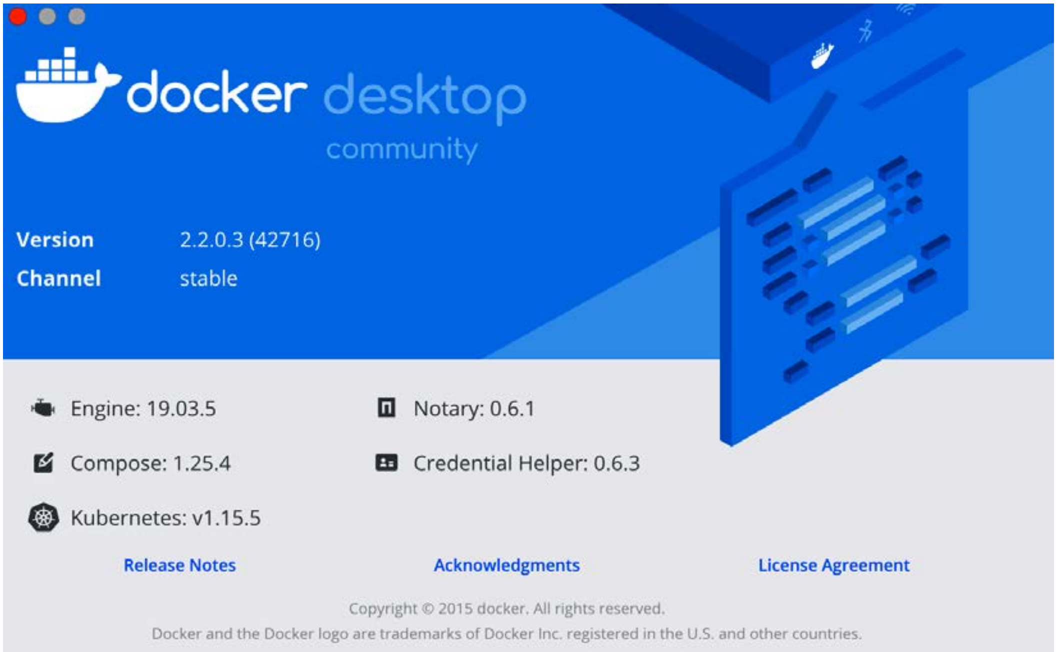
Docker Desktop needs privileged access to install its networking components and links to the Docker apps.

You will be asked for your password.

Figure 1.5 – Prompt for the Docker installer

6. Clicking on **OK** will bring up a dialogue that asks for your password. Once the password is entered, you should see a Docker icon in the top-left icon bar on your screen.

7. Clicking on the icon and selecting **About Docker** should show you something similar to the following:



Version 2.2.0.3 (42716)
Channel stable

Engine: 19.03.5
Notary: 0.6.1
Compose: 1.25.4
Credential Helper: 0.6.3
Kubernetes: v1.15.5

Copyright © 2015 docker. All rights reserved.
Docker and the Docker logo are trademarks of Docker Inc. registered in the U.S. and other countries.

Figure 1.6 – The About Docker screen

8. You can also run the following commands to check the version of Docker Compose that were installed alongside Docker Engine on the command line:

```
$ docker-compose version
```

Now that we know how to install Docker on macOS, let's move on to our final operating system, Windows 10 Professional.

Installing Docker on Windows 10 Professional

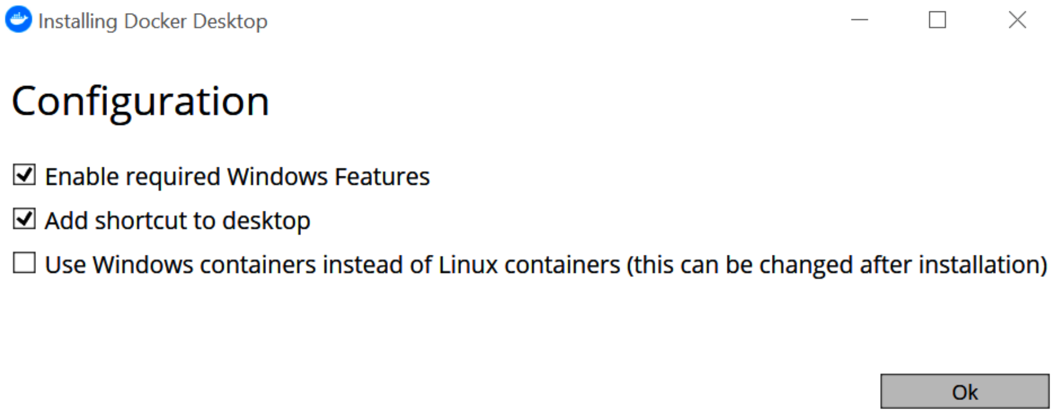
Like Docker for Mac, Docker for Windows uses a graphical installer.

Important Note

Before downloading, you should make sure that you are running Microsoft Windows 10 Professional or Enterprise 64-bit. If you are running an older version or an unsupported edition of Windows 10, you can still run Docker; refer to the *Older operating systems* section of this chapter for more information. Docker for Windows has this requirement due to its reliance on Hyper-V. Hyper-V is Windows' native hypervisor and allows you to run x86-64 guests on your Windows machine, be it Windows 10 Professional or Windows Server. It even forms part of the Xbox One operating system.

Let's install Docker for Windows:

1. Download the Docker for Windows installer from the Docker store at `https://hub.docker.com/editions/community/docker-ce-desktop-windows`.
2. Click on the **Get Docker** button to download the installer.
3. Once it's downloaded, run the installer package, and you will be greeted with the following:



Installing Docker Desktop

Configuration

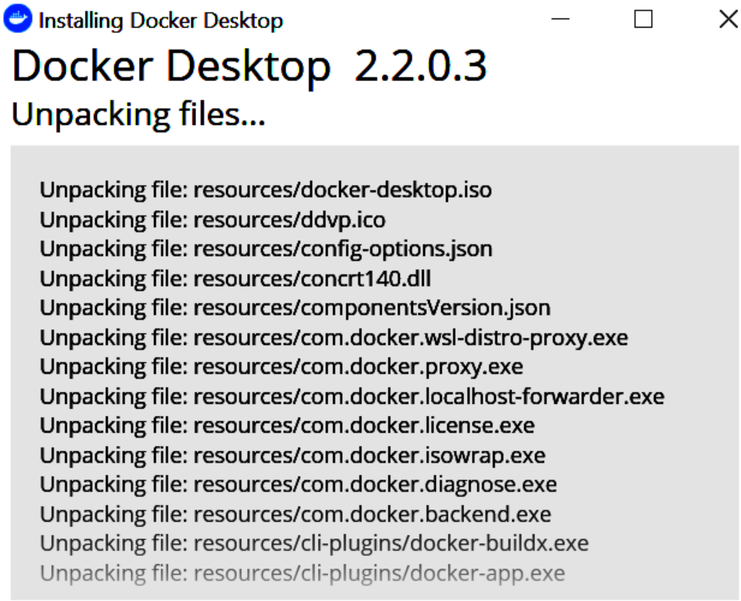
☑ Enable required Windows Features

☑ Add shortcut to desktop

□ Use Windows containers instead of Linux containers (this can be changed after installation)

Figure 1.7 – Configuration screen of the Docker for Windows installer

4. Leave the configuration at the default values and then click on **OK**. This will trigger an installation of all of the components needed to run Docker on Windows:



Installing Docker Desktop

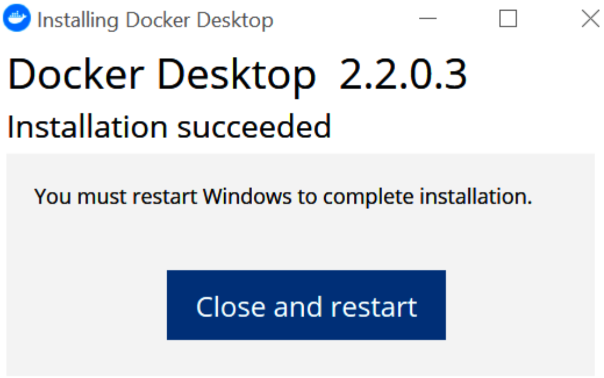
Docker Desktop 2.2.0.3

Unpacking files...

```
Unpacking file: resources/docker-desktop.iso
Unpacking file: resources/ddvp.ico
Unpacking file: resources/config-options.json
Unpacking file: resources/concrt140.dll
Unpacking file: resources/componentsVersion.json
Unpacking file: resources/com.docker.wsl-distro-proxy.exe
Unpacking file: resources/com.docker.proxy.exe
Unpacking file: resources/com.docker.localhost-forwarder.exe
Unpacking file: resources/com.docker.license.exe
Unpacking file: resources/com.docker.isowrap.exe
Unpacking file: resources/com.docker.diagnose.exe
Unpacking file: resources/com.docker.backend.exe
Unpacking file: resources/cli-plugins/docker-buildx.exe
Unpacking file: resources/cli-plugins/docker-app.exe
```

Figure 1.8 – Docker installation in progress

5. Once it's installed, you will be prompted to restart. To do this, simply click on the **Close and restart** button:



Installing Docker Desktop

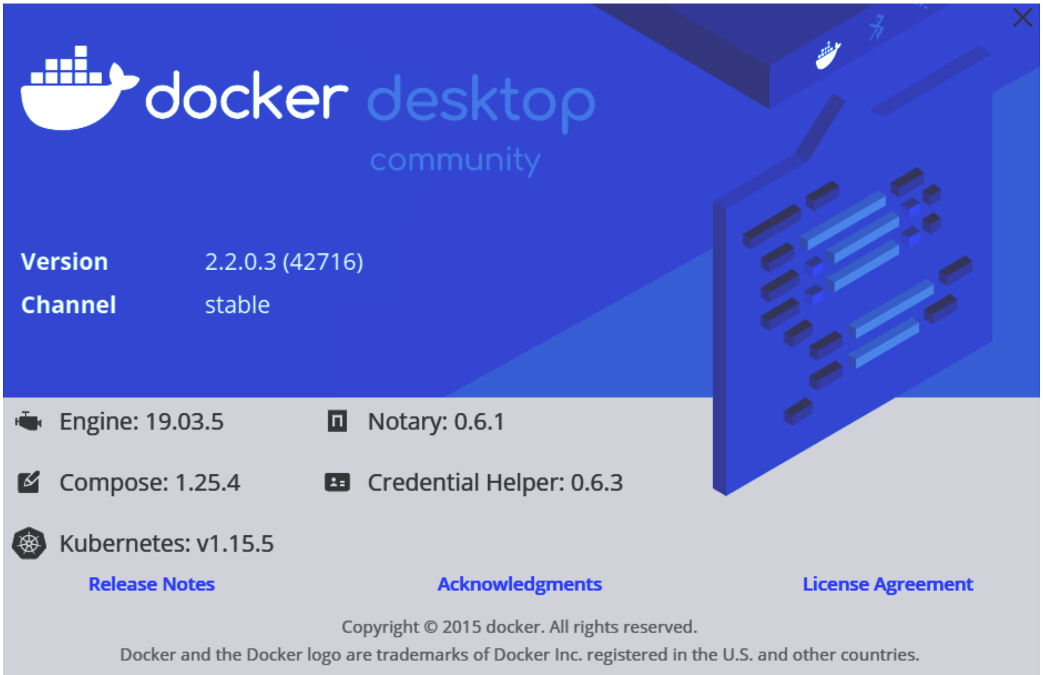
Docker Desktop 2.2.0.3

Installation succeeded

You must restart Windows to complete installation.

Figure 1.9 – Docker installation complete confirmation screen

6. Once your machine has restarted, you should see a Docker icon in the icon tray in the bottom right of your screen. Clicking on it and selecting **About Docker** from the menu will show the following:



Version 2.2.0.3 (42716)

Channel stable

Engine: 19.03.5

Notary: 0.6.1

Compose: 1.25.4

Credential Helper: 0.6.3

Kubernetes: v1.15.5

Copyright © 2015 docker. All rights reserved.

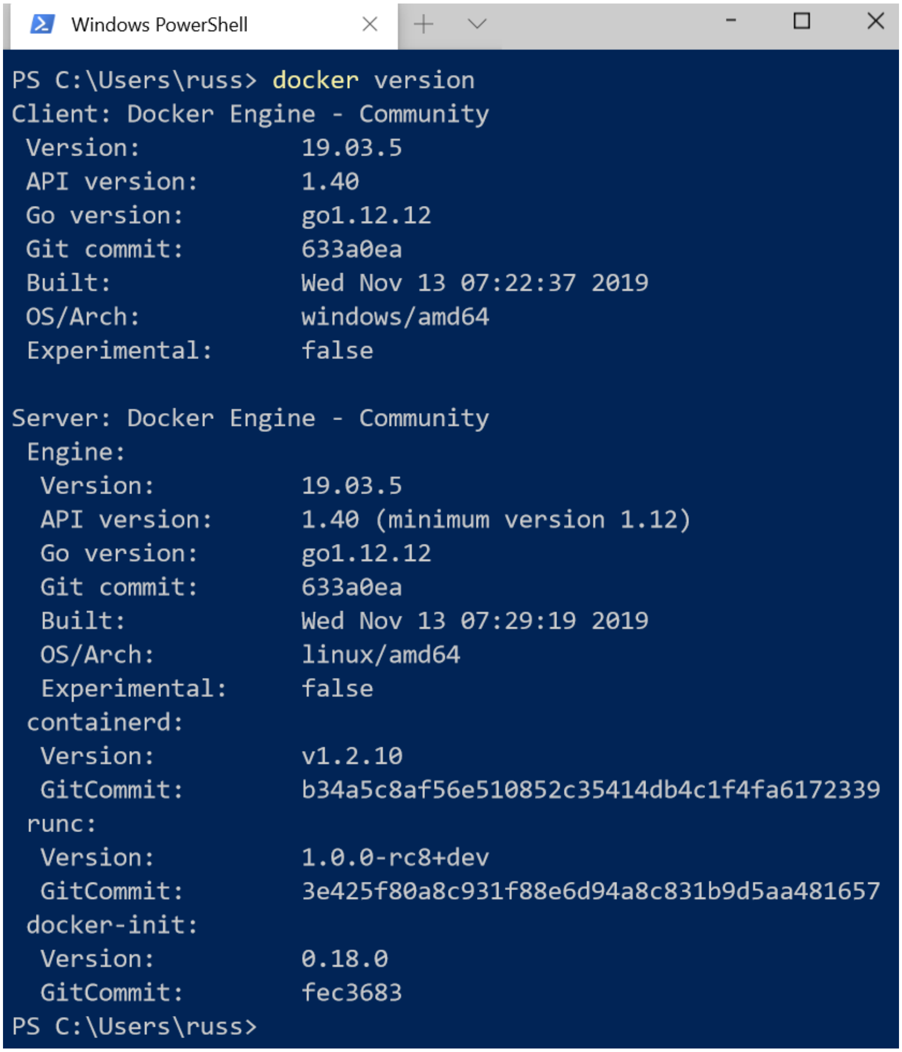
Docker and the Docker logo are trademarks of Docker Inc. registered in the U.S. and other countries.

Figure 1.10 – Docker About Me page

7. Open a PowerShell window and type the following command:

```
$ docker version
```

This should also show you similar output to the Mac and Linux versions:



```
PS C:\Users\russ> docker version
Client: Docker Engine - Community
 Version:           19.03.5
 API version:       1.40
 Go version:        go1.12.12
 Git commit:        633a0ea
 Built:             Wed Nov 13 07:22:37 2019
 OS/Arch:           windows/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.5
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.12
  Git commit:       633a0ea
  Built:            Wed Nov 13 07:29:19 2019
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          v1.2.10
  GitCommit:        b34a5c8af56e510852c35414db4c1f4fa6172339
 runc:
  Version:          1.0.0-rc8+dev
  GitCommit:        3e425f80a8c931f88e6d94a8c831b9d5aa481657
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
PS C:\Users\russ>
```

Figure 1.11 – Output of the docker version command

Again, you can also run the following commands to check the versions of Docker Compose and Docker Machine that were installed alongside Docker Engine:

```
$ docker-compose version
```

You should see a similar output to the macOS and Linux versions. As you may have started to gather, once the packages are installed, their usage is going to be pretty similar. You will be able to see this when we get to the *Using Docker commands* section of this chapter.

Older operating systems

If you are not running a sufficiently new operating system on Mac or Windows, then you will need to use Docker Toolbox. Consider the output printed from running the following command:

```
$ docker version
```

On all three of the installations we have performed so far, it shows two different versions, a client and a server. Predictably, the Linux version shows that the architecture for the client and server are both Linux; however, you may notice that the Mac version shows the client is running on Darwin, which is Apple's Unix-like kernel, and the Windows version shows Windows. Yet both of the servers show the architecture as being Linux, so what gives?

That is because both the Mac and Windows versions of Docker download and run a virtual machine in the background, and this virtual machine runs a small, lightweight operating system based on Alpine Linux. The virtual machine runs using Docker's libraries, which connect to the built-in hypervisor for your chosen environment.

For macOS, this is the built-in **Hypervisor.framework**, and for Windows, as we have already mentioned, it is **Hyper-V**.

To ensure that no one misses out on the Docker experience, a version of Docker that does not use these built-in hypervisors is available for older versions of macOS and unsupported Windows versions. These versions utilize **VirtualBox** as the hypervisor to run the Linux server for your local client to connect to.

Important note

VirtualBox is an open source x86 and AMD64/Intel64 virtualization product developed by Oracle. It runs on Windows, Linux, Macintosh, and Solaris hosts, with support for many Linux, Unix, and Windows guest operating systems.

For more information on Docker Toolbox, see the project's website at `https://github.com/docker/toolbox/`, where you can also download the macOS and Windows installers from the releases page.

Important note

This book assumes that you have installed the latest Docker version on Linux or have used Docker for Mac or Docker for Windows. While Docker installations using Docker Toolbox should be able to support the commands in this book, you may run into issues around file permissions and ownership when mounting data from your local machine to your containers.

Now that you have Docker up and running on the system of your choice, let's start exploring the commands that we need in order to use it effectively.

Using Docker commands

You should already be familiar with these Docker commands. However, it's worth going through them to ensure you know all. We will start with some common commands and then take a peek at the commands that are used for the Docker images. We will then take a dive into the commands that are used for the containers.

Tip

A while ago, Docker restructured their command-line client into more logical groupings of commands, as the number of features provided by the client multiplies and commands start to cross over each other. Throughout this book, we will be using this structure rather than some of the shorthand that still exists within the client.

The first command we will be taking a look at is one of the most useful commands, not only in Docker but in any command-line utility you use – the `help` command. It is run simply like this:

```
$ docker help
```

This command will give you a full list of all of the Docker commands at your disposal, along with a brief description of what each command does. We will be looking at this in more detail in *Chapter 4, Managing Containers*. For further help with a particular command, you can run the following:

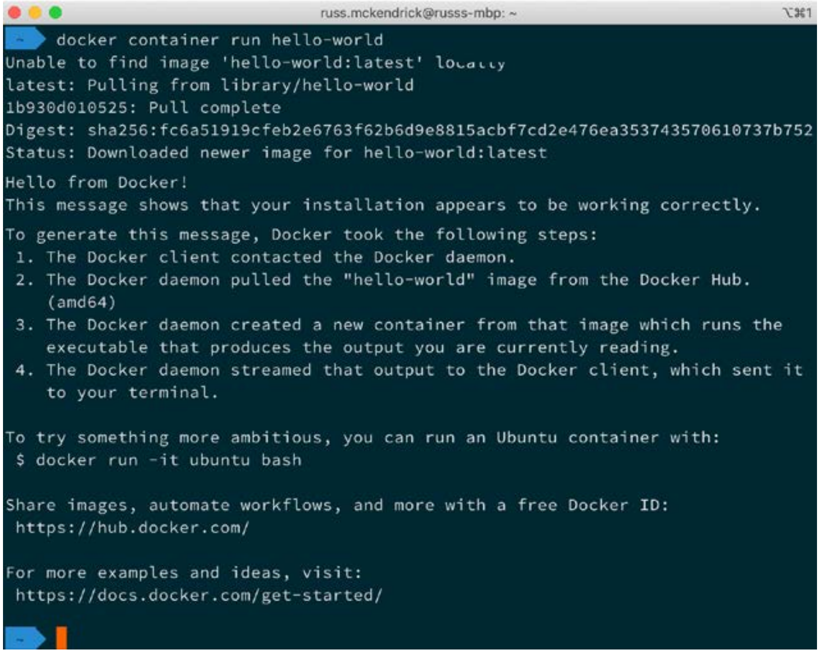
```
$ docker <COMMAND> --help
```

Next, let's run the `hello-world` container. To do this, simply run the following command:

```
$ docker container run hello-world
```

It doesn't matter what host you are running Docker on, the same thing will happen on Linux, macOS, and Windows. Docker will download the `hello-world` container image and then execute it, and once it's executed, the container will be stopped.

Your Terminal session should look like the following:



```
~  docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:fc6a51919cfeb2e6763f62b6d9e8815acbf7cd2e476ea353743570610737b752
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

~ 
```

Figure 1.12 – Output for docker container run hello-world

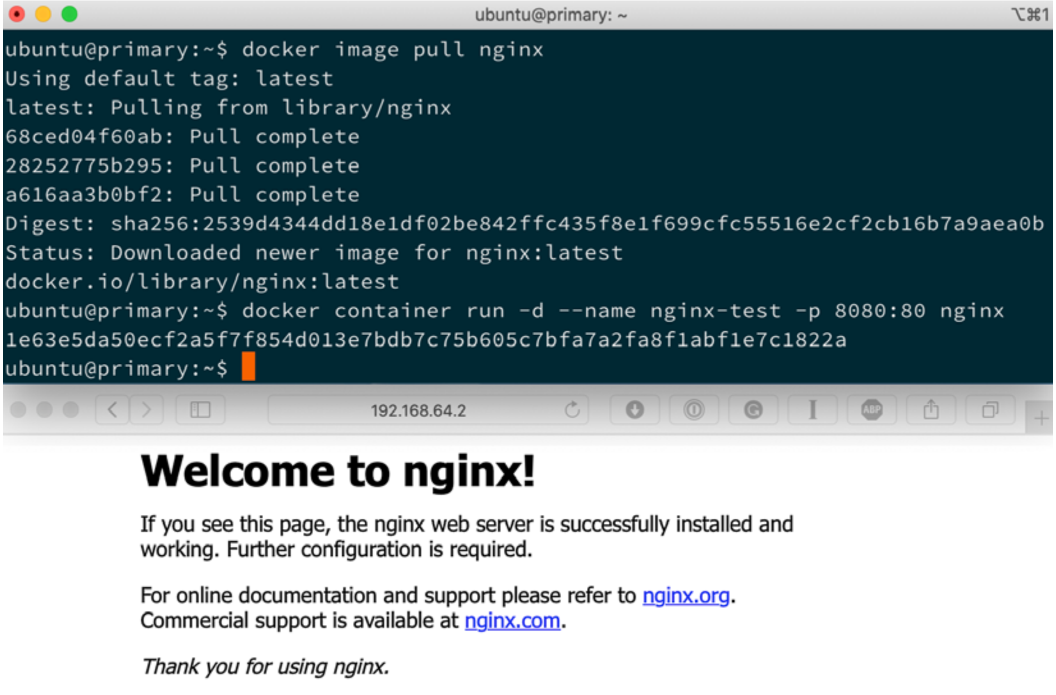
Let's try something a little more adventurous – let's download and run an NGINX container by running the following two commands:

```
$ docker image pull nginx
$ docker container run -d --name nginx-test -p 8080:80 nginx
```

> **Important note**
>
> NGINX is an open source web server that can be used as a load balancer, mail proxy, reverse proxy, and even an HTTP cache.

The first of the two commands downloads the NGINX container image, and the second command launches a container in the background called `nginx-test`, using the `nginx` image we pulled. It also maps port `8080` on our host machine to port `80` on the container, making it accessible to our local browser at `http://localhost:8080/`.

As you can see from the following screenshots, the command and results are exactly the same on all three OS types. Here we have Linux:



```
ubuntu@primary:~$ docker image pull nginx
Using default tag: latest
latest: Pulling from library/nginx
68ced04f60ab: Pull complete
28252775b295: Pull complete
a616aa3b0bf2: Pull complete
Digest: sha256:2539d4344dd18e1df02be842ffc435f8e1f699cfc55516e2cf2cb16b7a9aea0b
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
ubuntu@primary:~$ docker container run -d --name nginx-test -p 8080:80 nginx
1e63e5da50ecf2a5f7f854d013e7bdb7c75b605c7bfa7a2fa8f1abf1e7c1822a
ubuntu@primary:~$ 
```

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

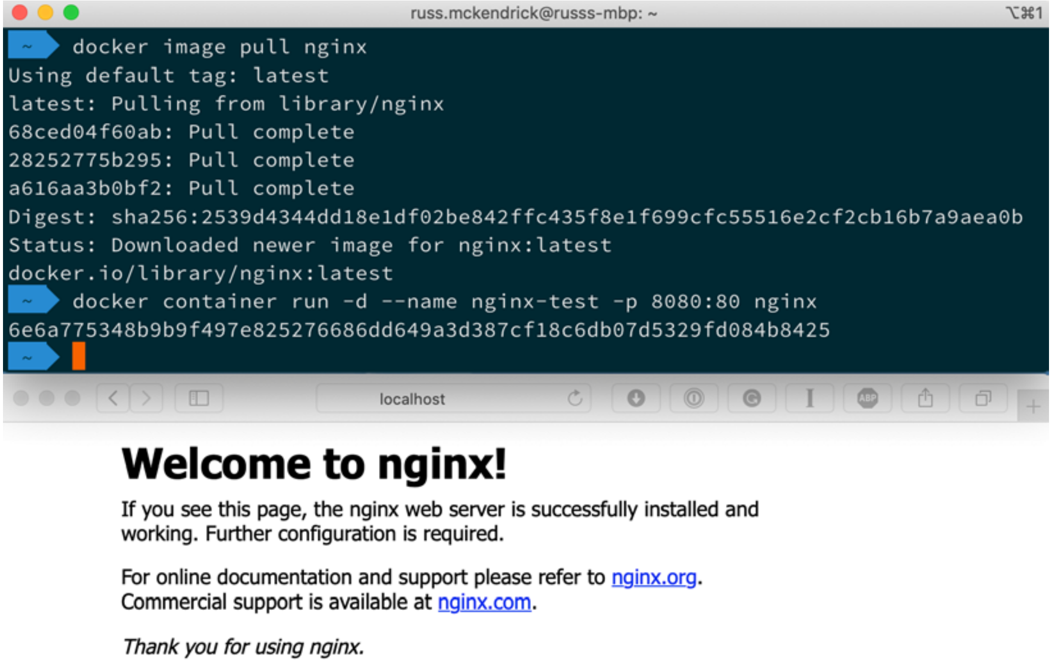
For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 1.13 – Output of docker image pull nginx on Linux

> **Important note**
>
> You may notice that the Linux and macOS screens at first glance look similar. That is because I am using a remote Linux server, and we will look more at how to do this in a later chapter.

This is the result on macOS:



```
~ docker image pull nginx
Using default tag: latest
latest: Pulling from library/nginx
68ced04f60ab: Pull complete
28252775b295: Pull complete
a616aa3b0bf2: Pull complete
Digest: sha256:2539d4344dd18e1df02be842ffc435f8e1f699cfc55516e2cf2cb16b7a9aea0b
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
~ docker container run -d --name nginx-test -p 8080:80 nginx
6e6a775348b9b9f497e825276686dd649a3d387cf18c6db07d5329fd084b8425
~
```

Welcome to nginx!

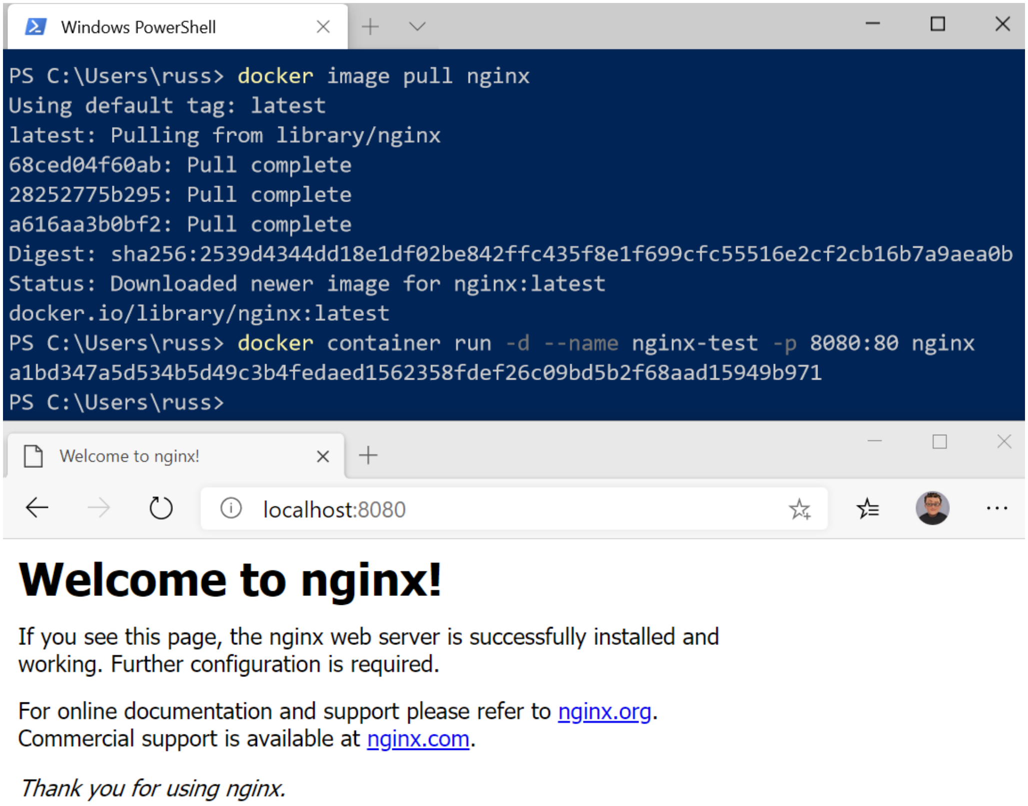
If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 1.14 – Output of docker image pull nginx on macOS

And this is how it looks on Windows:



```
PS C:\Users\russ> docker image pull nginx
Using default tag: latest
latest: Pulling from library/nginx
68ced04f60ab: Pull complete
28252775b295: Pull complete
a616aa3b0bf2: Pull complete
Digest: sha256:2539d4344dd18e1df02be842ffc435f8e1f699cfc55516e2cf2cb16b7a9aea0b
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
PS C:\Users\russ> docker container run -d --name nginx-test -p 8080:80 nginx
a1bd347a5d534b5d49c3b4fedaed1562358fdef26c09bd5b2f68aad15949b971
PS C:\Users\russ>
```

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 1.15 – Output of docker image pull nginx on Windows

In the following three chapters, we will look at using the Docker command-line client in more detail. For now, let's stop and remove our `nginx-test` container by running the following:

```
$ docker container stop nginx-test
$ docker container rm nginx-test
```

As you can see, the experience of running a simple NGINX container on all three of the hosts on which we have installed Docker is exactly the same. As am I sure you can imagine, trying to achieve this without something like Docker across all three platforms is a challenge, and a very different experience on each platform too. Traditionally, this has been one of the reasons for the difference in local development environments as people would need to download a platform-specific installer and configure the service for the platform they are running. Also, in some cases there could be feature differences between the platforms.

Now that we have a foundation in Docker commands, let's cast a wider net and look at its container ecosystem.

Docker and the container ecosystem

If you have been following the rise of Docker and containers, you will have noticed that, throughout the last few years, the messaging on the Docker website has been slowly changing from headlines about what containers are to more of a focus on the services provided by Docker as a company.

One of the core drivers for this is that everything has traditionally been lumped into being known just as 'Docker,' which can get confusing. Now that people did not need educating as much on what a container is or the problems they can solve with Docker, the company needed to try and start to differentiate themselves from competitors that sprung up to support all sorts of container technologies.

So, let's try and unpack everything that is Docker, which involves the following:

- **Open source projects**: There are several open source projects started by Docker, which are now maintained by a large community of developers.
- **Docker, Inc**.: This is the company founded to support and develop the core Docker tools.
- **Docker CE and Docker EE**: This is the core collection of Docker tools built on top of the open source components.

We will also be looking at some third-party services in later chapters. In the meantime, let's go into more detail on each of these, starting with the open source projects.

Open source projects

Docker, Inc. has spent the last few years open sourcing and donating a lot of its core projects to various open source foundations and communities. These projects include the following:

- **Moby Project** is the upstream project upon which the Docker Engine is based. It provides all of the components needed to assemble a fully functional container system.
- **Runc** is a command-line interface for creating and configuring containers and has been built to the OCI specification.
- **Containerd** is an easily embeddable container runtime. It is also a core component of the Moby Project.

- **LibNetwork** is a Go library that provides networking for containers. Notary is a client and server that aims to provide a trust system for signed container images.
- **HyperKit** is a toolkit that allows you to embed hypervisor capabilities into your own applications; presently, it only supports the macOS and the Hypervisor framework.
- **VPNKit** provides VPN functionality to HyperKit.
- **DataKit** allows you to orchestrate application data using a Git-like workflow.
- **SwarmKit** is a toolkit that enables you to build distributed systems using the same raft consensus algorithm as Docker Swarm.
- **LinuxKit** is a framework that allows you to develop and compile a small portable Linux operating system for running containers.
- **InfraKit** is a collection of tools that you can use to define the infrastructure to run your **LinuxKit** generated distributions on.

On their own, you will probably never use the individual components; however, each of the projects mentioned is a component of the tools that are maintained by Docker, Inc. We will go a little more into these projects in our final chapter.

Docker, Inc.

Docker, Inc. is the company formed to initially develop **Docker Community Edition** (**Docker CE**) and **Docker Enterprise Edition** (**Docker EE**). It also used to provide an SLA-based support service for Docker EE as well as offering consulting services to companies who wish to take their existing applications and containerize them as part of Docker's **Modernise Traditional Apps** (**MTA**) program.

You will notice that I referred to a lot of the things in the previous sentence in the past tense. This is because in November 2019 Docker, Inc. restructured and sold its platform business to a company called Mirantis Inc. They acquired the following assets from Docker, Inc.:

- **Docker Enterprise**, including Docker EE
- **Docker Trusted Registry**
- **Docker Unified Control Plane**
- **Docker CLI**

Mirantis Inc. is a California-based company that focuses on the development and support of **OpenStack**- and **Kubernetes-based** solutions. It was one of the founders of the non-profit corporate entity OpenStack Foundation and had a vast amount of experience of providing enterprise-level support.

Former Docker, Inc. CEO Rob Bearden, who stepped down shortly after the announcement, was quoted as saying:

> *'After conducting thorough analysis with the management team and the Board of Directors, we determined that Docker had two very distinct and different businesses: one an active developer business, and the other a growing enterprise business. We also found that the product and the financial models were vastly different. This led to the decision to restructure the company and separate the two businesses, which is the best thing for customers and to enable Docker's industry-leading technology to thrive.'*

With the Enterprise business now with Mirantis Inc., Docker, Inc. is focusing on providing better developer workflows with Docker Desktop and Docker Hub, which allows users to avoid the threat of vendor lock-in.

Docker CE and Docker EE

There are a lot of tools supplied and supported by Docker, Inc. Some we have already mentioned, and others we will cover in later chapters. Before we finish this, our first chapter, we should get an idea of the tools we are going to be using. The most of important of them is the core Docker Engine.

This is the core of Docker, and all of the other tools that we will be covering use it. We have already been using it as we installed it in the Docker installation and Docker commands sections of this chapter. There are currently two versions of Docker Engine; there is Docker EE, which is now maintained by Mirantis Inc., and Docker CE. We will be using Docker CE throughout this book.

As well as the stable version of Docker CE, Docker will be providing nightly builds of the Docker Engine via a nightly repository (formally Docker CE Edge), and monthly builds of Docker for Mac and Docker for Windows via the Edge channel.

There are also the following tools:

- **Docker Compose**: A tool that allows you to define and share multi-container definitions; it is detailed in *Chapter 5*, *Docker Compose*.
- **Docker Machine**: A tool to launch Docker hosts on multiple platforms; we will cover this in *Chapter 6*, *Managing Containers*.

- **Docker Hub**: A repository for your Docker images, covered in the next three chapters.
- **Docker Desktop (Mac)**: We have covered Docker for Mac in this chapter.
- **Docker Desk/top (Windows)**: We have covered Docker for Windows in this chapter.
- **Docker Swarm**: A multi-host-aware orchestration tool, covered in detail in *Chapter 8*, *Docker Swarm*. Mirantis Inc now maintains this.

Summary

In this chapter, we covered some basic information that you should already know (or now know) for the chapters ahead. We went over the basics of what Docker is, and how it fares compared to other host types. We went over the installers, how they operate on different operating systems, and how to control them through the command line. Be sure to remember to look at the requirements for the installers to ensure you use the correct one for your operating system.

Then, we took a small dive into using Docker and issued a few basic commands to get you started. We will be looking at all of the management commands in future chapters to get a more in-depth understanding of what they are, as well as how and when to use them. Finally, we discussed the Docker ecosystem and the responsibilities of each of the different tools.

In the next chapter, we will be taking a look at how to build base containers, and we will also look in depth at Dockerfiles and places to store your images, as well as using environmental variables and Docker volumes.

Questions

1. Where can you download Docker Desktop (Mac) and Docker Desktop (Windows) from?
2. What command did we use to download the NGINX image?
3. Which open source project is upstream for the core Docker Engine?
4. Which company now maintains Docker Enterprise?
5. Which command would you run to find out more information on the Docker container subset of commands?

Further reading

These are the companies involved in maintaining Docker:

- Docker, Inc.: `http://docker.com`
- Mirantis Inc.: `https://www.mirantis.com`
- Docker restructure: `https://www.computerweekly.com/news/252473956/Docker-restructure-sees-enterprise-platform-business-sold-to-open-source-cloud-firm-Mirantis`

In this chapter, we have mentioned the following hypervisors:

- macOS Hypervisor framework: `https://developer.apple.com/documentation/hypervisor`
- Hyper-V: `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v`

For details on how to install on other Linux distributions, take a look at the Install Docker Engine page of the Docker docs: `https://docs.docker.com/engine/install/`.

We referenced the following blog posts from Docker:

- Docker CLI restructure blog post: `https://www.docker.com/blog/whats-new-in-docker-1-13/`
- Docker Extended Support Announcement: `https://www.docker.com/blog/extending-support-cycle-docker-community-edition/`

Next up, we discussed the following open source projects:

- Moby Project: `https://mobyproject.org`
- Runc: `https://github.com/opencontainers/runc`
- Containerd: `https://containerd.io`
- LibNetwork: `https://github.com/moby/libnetwork`
- Notary: `https://github.com/theupdateframework/notary`
- HyperKit: `https://github.com/moby/hyperkit`
- VPNKit: `https://github.com/moby/vpnkit`
- DataKit: `https://github.com/moby/datakit`

2
Building Container Images

In this chapter, you will start building container images. We will look at five different ways you can define and build images using native Docker tools.

We will discuss the recommended ways that you can define and build your own images, as well as one way that is not considered to be a best practice but does have its uses.

We will cover the following topics:

- Introducing Dockerfiles
- Building Docker images

Let's get started!

Technical requirements

In this chapter, we will be using our Docker installation to build images. Some of the supporting commands, which will be few and far between, may only be applicable to macOS and Linux-based operating systems.

Check out the following video to see the Code in Action:
`https://bit.ly/3h7oDX5`

Tip

While the screenshots in this chapter will be from my preferred operating system, which is macOS, the Docker commands we will be running will work on all three operating systems we have installed Docker on so far.

Introducing Dockerfiles

In this section, we will cover Dockerfiles in depth, along with the best practices when it comes to their use. So, what is a Dockerfile?

A **Dockerfile** is simply a plain text file that contains a set of user-defined instructions. When a Dockerfile is called by the `docker image build` command, which we will look at next, it is used to assemble a container image.

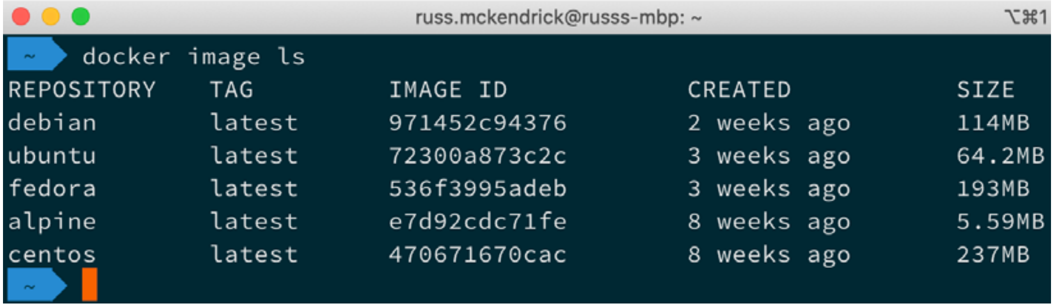
A Dockerfile looks as follows:

```
FROM alpine:latest
LABEL maintainer=”Russ McKendrick <russ@mckendrick.io>”
LABEL description=”This example Dockerfile installs NGINX.”
RUN apk add --update nginx && \
    rm -rf /var/cache/apk/* && \
    mkdir -p /tmp/nginx/
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/default.conf /etc/nginx/conf.d/default.conf
ADD files/html.tar.gz /usr/share/nginx/

EXPOSE 80/tcp

ENTRYPOINT [“nginx”]
CMD [“-g”, “daemon off;”]
```

As you can see, even with no explanation, it is quite easy to get an idea of what each step of the Dockerfile instructs the build command to do. Before we move on and work our way through the previous file, we should quickly touch upon Alpine Linux.

Alpine Linux is a small, independently developed, non-commercial Linux distribution designed for security, efficiency, and ease of use. While small, it offers a solid foundation for container images due to its extensive repository of packages, and also thanks to the unofficial port of `grsecurity/PaX`, which is patched into its kernel. This port offers proactive protection against dozens of potential zero-day threats and other vulnerabilities.

Alpine Linux, due to both its size and how powerful it is, has become the default image base for the official container images supplied by Docker. Because of this, we will be using it throughout this book. To give you an idea of just how small the official image for Alpine Linux is, let's compare it to some of the other distributions available at the time of writing:



```
~ docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
debian        latest    971452c94376   2 weeks ago    114MB
ubuntu        latest    72300a873c2c   3 weeks ago    64.2MB
fedora        latest    536f3995adeb   3 weeks ago    193MB
alpine        latest    e7d92cdc71fe   8 weeks ago    5.59MB
centos        latest    470671670cac   8 weeks ago    237MB
~
```

Figure 2.1 – Comparing the size of popular base images

As you can see from the preceding Terminal output, Alpine Linux weighs in at only 5.59 MB, as opposed to the biggest image, which is CentOS, at 237 MB. A bare-metal installation of Alpine Linux comes in at around 130 MB, which is still almost half the size of the CentOS container image.

Reviewing Dockerfiles in depth

Let's take a look at the instructions we used in the preceding Dockerfile example. We will look at them in the order they appeared in:

- `FROM`
- `LABEL`
- `RUN`
- `COPY` and `ADD`
- `EXPOSE`
- `ENTRYPOINT` and `CMD`
- Other Dockerfile instructions

FROM

The `FROM` instruction tells Docker which base you would like to use for your image. As we already mentioned, we are using Alpine Linux, so we simply have to state the name of the image and the release tag we wish to use. In our case, to use the latest official Alpine Linux image, we simply need to add `alpine:latest`.

LABEL

The `LABEL` instruction can be used to add extra information to the image. This information can be anything from a version number to a description. It's also recommended that you limit the number of labels you use. A good label structure will help others who will use our image later on.

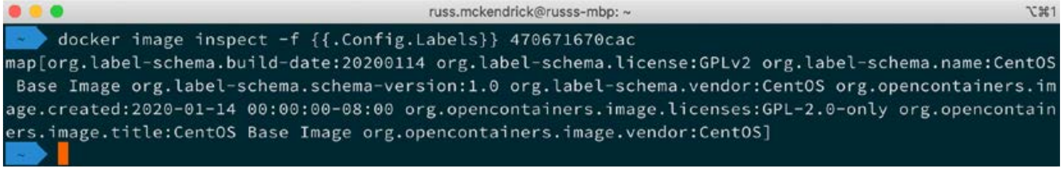
However, using too many labels can cause the image to become inefficient as well, so I would recommend using the label schema detailed at `http://label-schema.org`. You can view the containers' labels with the following `docker inspect` command:

```
$ docker image inspect <IMAGE_ID>
```

Alternatively, you can use the following command to filter just the labels:

```
$ docker image inspect -f {{.Config.Labels}} <IMAGE_ID>
```

In the following screenshot, you can see the labels present for the CentOS image:



```
~ docker image inspect -f {{.Config.Labels}} 470671670cac
map[org.label-schema.build-date:20200114 org.label-schema.license:GPLv2 org.label-schema.name:CentOS
 Base Image org.label-schema.schema-version:1.0 org.label-schema.vendor:CentOS org.opencontainers.im
age.created:2020-01-14 00:00:00-08:00 org.opencontainers.image.licenses:GPL-2.0-only org.opencontain
ers.image.title:CentOS Base Image org.opencontainers.image.vendor:CentOS]
~
```

Figure 2.2 – Checking image label

In our example Dockerfile, we have added two labels:

`maintainer="Russ McKendrick <russ@mckendrick.io>"`: Adds a label that helps the end user of the image identify who is maintaining it.

`description="This example Dockerfile installs NGINX."`: Adds a brief description of what the image is.

Generally, it is better to define your labels when you create a container from your image, rather than at build time, so it is best to keep labels down to just metadata about the image and nothing else.

RUN

The RUN instruction is where we interact with our image to install software and run scripts, commands, and other tasks. As you can see from the following RUN instruction, we are actually running three commands:

```
RUN apk add --update nginx && \
    rm -rf /var/cache/apk/* && \
    mkdir -p /tmp/nginx/
```

The first of our three commands is the equivalent of running the following command if we had a shell on an Alpine Linux host:

```
$ apk add --update nginx
```

This command installs NGINX using Alpine Linux's package manager.

> **Tip**
>
> We are using the && operator to move on to the next command if the previous command was successful. This makes it more obvious which commands we are running in the Dockerfile. We are also using \, which allows us to split the command over multiple lines, making it even easier to read.

The following command in our chain removes any temporary files to keep the size of our image to a minimum:

```
$ rm -rf /var/cache/apk/*
```

The final command in our chain creates a folder with a path of /tmp/nginx/ so that NGINX will start correctly when we run the container:

```
$ mkdir -p /tmp/nginx/
```

We could have also used the following in our Dockerfile to achieve the same results:

```
RUN apk add --update nginx
RUN rm -rf /var/cache/apk/*
RUN mkdir -p /tmp/nginx/
```

However, much like adding multiple labels, this is considered inefficient as it can add to the overall size of the image, which we should try to avoid. There are some valid use cases for this as some commands do not work well when they are stringed together using `&&`. However, for the most part, this approach to running commands should be avoided when your image is being built.

COPY and ADD

At first glance, `COPY` and `ADD` look like they are doing the same task in that they are both used to transfer files to the image. However, there are some important differences, which we will discuss here.

The `COPY` instruction is the more straightforward of the two:

```
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/default.conf /etc/nginx/conf.d/default.conf
```

As you have probably guessed, we are copying two files from the `files` folder on the host we are building our image on. The first file is `nginx.conf`, which is a minimal NGINX configuration file:

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
    log_format  main  '$remote_addr - $remote_user [$time_
local] "$request" '
                      '$status $body_bytes_sent "$http_referer"
'
                      '"$http_user_agent" "$http_x_forwarded_
for"';
```

```
    access_log  /var/log/nginx/access.log  main;
    sendfile        off;
    keepalive_timeout  65;
    include /etc/nginx/conf.d/*.conf;
}
```

This will overwrite the NGINX configuration that was installed as part of the APK installation in the `RUN` instruction.

The next file, `default.conf`, is the simplest virtual host that we can configure, and contains the following content:

```
server {
  location / {
      root /usr/share/nginx/html;
  }
}
```

Again, this will overwrite any existing files. So far, so good! So, why might we use the `ADD` instruction?

In our example Dockerfile, the `ADD` instruction looks as follows:

```
ADD files/html.tar.gz /usr/share/nginx/
```

As you can see, we are adding a file called `html.tar.gz`, but we are not actually doing anything with the archive to uncompress it in our Dockerfile. This is because `ADD` automatically uploads, uncompresses, and adds the resulting folders and files to the path we request it to, which in our case is `/usr/share/nginx/`. This gives us our web root of `/usr/share/nginx/html/`, as we defined in the virtual host block in the `default.conf` file that we copied to the image.

The `ADD` instruction can also be used to add content from remote sources. For example, consider the following:

```
ADD https://raw.githubusercontent.com/PacktPublishing/
Mastering-Docker-Fourth-Edition/master/chapter02/dockerfile-
example/files/html.tar.gz /usr/share/nginx/
```

The preceding command line would download `html.tar.gz` from `https://raw.githubusercontent.com/PacktPublishing/Mastering-Docker-Fourth-Edition/master/chapter02/dockerfile-example/files/` and place the file in the `/usr/share/nginx/` folder on the image.

Archive files from a remote source are treated as files and are not uncompressed, which you will have to take into account when using them. This means that the file will have to be added before the `RUN` instruction so that we can manually unarchive the folder and also remove the `html.tar.gz` file.

`EXPOSE`: The `EXPOSE` instruction lets Docker know that when the image is executed, the port and protocol defined will be exposed at runtime. This instruction does not map the port to the host machine; instead, it opens the port to allow access to the service on the container network.

For example, in our Dockerfile, we are telling Docker to open port `80` every time the image runs:

```
EXPOSE 80/tcp
```

The benefit of using `ENTRYPOINT` over `CMD` is that you can use them in conjunction with each other. `ENTRYPOINT` can be used by itself but remember that you would only want to use `ENTRYPOINT` by itself if you wanted your container to be executable.

For reference, if you think of some of the `CLI` commands you might use, you must specify more than just the `CLI` command. You might have to add extra parameters that you want the command to interpret. This would be the use case for using `ENTRYPOINT` only.

For example, if you want to have a default command that you want to execute inside a container, you could do something similar to the following example. Be sure to use a command that keeps the container alive.

Here, we are using the following:

```
ENTRYPOINT [“nginx”]
CMD [“-g”, “daemon off;”]
```

What this means is that whenever we launch a container from our image, the NGINX binary is executed, which, as we have defined, is our entry point. Then, whatever we have as `CMD` is executed, giving us the equivalent of running the following command:

```
$ nginx -g daemon off;
```

Another example of how `ENTRYPOINT` can be used is as follows:

```
$ docker container run --name nginx-version dockerfile-example
-v
```

This would be the equivalent of running the following command on our host:

```
$ nginx -v
```

Notice that we didn't have to tell Docker to use NGINX. Since we have the NGINX binary as our entry point, any command we pass overrides the `CMD` instruction that has been defined in the Dockerfile.

This would display the version of NGINX we have installed and our container would stop, as the NGINX binary would only be executed to display the version information. We will look at this once we have built and launched a container using our image. Before we move on, we should look at some of the instructions that are not included in our Dockerfile.

Other Dockerfile instructions

There are some instructions that we have not included in our example Dockerfile. Let's take a look at them here:

- `USER`: The `USER` instruction lets you specify the username to be used when a command is run. The `USER` instruction can be used on the `RUN` instruction, the `CMD` instruction, or the `ENTRYPOINT` instruction in the Dockerfile. Also, the user defined in the `USER` instruction must exist, or your image will fail to build. Using the `USER` instruction can also introduce permission issues, not only on the container itself, but also if you mount volumes.

- WORKDIR: The WORKDIR instruction sets the working directory for the same set of instructions that the USER instruction can use (RUN, CMD, and ENTRYPOINT). It will allow you to use the CMD and ADD instructions as well.
- ONBUILD: The ONBUILD instruction lets you stash a set of commands to be used when the image is used in the future, as a base image for another container image. For example, if you want to give an image to developers and they all have a different code base that they want to test, you can use the ONBUILD instruction to lay the groundwork ahead of the fact of needing the actual code. Then, the developers will simply add their code to the directory you ask them to, and when they run a new Docker build command, it will add their code to the running image.
  The ONBUILD instruction can be used in conjunction with the ADD and RUN instructions, such as in the following example:

```
ONBUILD RUN apk update && apk upgrade && rm -rf /var/cache/
apk/*
```

  This would run an update and package upgrade every time our image is used as a base for another container image.
- ENV: The ENV instruction sets ENVs within the image both when it is built and when it is executed. These variables can be overridden when you launch your image.

Dockerfiles – best practices

Now that we have covered Dockerfile instructions, let's take a look at a few tips that are considered best practices for writing our own Dockerfiles. Following these will ensure that your images are lean, consistent, and easy for others to follow:

- You should try to get into the habit of using a .dockerignore file. We will cover the .dockerignore file in the *Building Docker images* section of this chapter; it will seem very familiar if you are used to using a .gitignore file. It will essentially ignore the items you specified in the file during the build process.
- Remember to only have one Dockerfile per folder to help you organize your containers.

- Use a version control system, such as Git, for your Dockerfile; just like any other text-based document, version control will help you move not only forward, but also backward, as necessary.
- Minimize the number of packages you need to install per image. One of the biggest goals you want to achieve while building your images is to keep them as small and secure as possible. Not installing unnecessary packages will greatly help in achieving this goal.
- Make sure there is only one application process per container. Every time you need a new application process, it is good practice to use a new container to run that application in.
- Keep things simple; over-complicating your Dockerfile will add bloat and potentially cause you issues down the line.
- Learn by example! Docker themselves have quite a detailed style guide for publishing the official images they host on Docker Hub. You can find a link to this in the *Further reading* section at the end of this chapter.

Building Docker images

In this section, we will cover the `docker image build` command. This is where the rubber meets the road, as they say. It's time for us to build the base upon which we will start building our future images. We will be looking at different ways to accomplish this goal. Consider this as a template that you may have created earlier with virtual machines. This will help save you time as this will complete the hard work for you; you will just have to create the application that needs to be added to the new images.

There are a lot of switches that you can use while using the `docker build` command. So, let's use the one that is always handy. Here, we will use the `--help` switch on the `docker image build` command to view what we can do:

```
$ docker image build --help
```

There are a lot of different flags listed that you can pass when building your image. Now, it may seem like a lot to digest, but out of all these options, we only need to use `--tag`, or its shorthand, `-t`, to name our image.

You can use the other options to limit how much CPU and memory the build process will use. In some cases, you may not want the `build` command to take as much CPU or memory as it can have. The process may run a little slower, but if you are running it on your local machine or a production server and it's a long build process, you may want to set a limit. There are also options that affect the network configuration of the container that was launched to build our image.

Typically, you don't use the `--file` or `-f` switch since you run the `docker build` command from the same folder that the Dockerfile is in. Keeping the Dockerfile in separate folders helps sort the files and keeps the naming convention of the files the same.

It's also worth mentioning that, while you are able to pass additional ENVs as arguments at build time, they are used at build time and your container image does not inherit them. This is useful for passing information such as proxy settings, which may only be applicable to your initial build/test environment.

The `.dockerignore` file, as we discussed earlier, is used to exclude those files or folders we don't want to be included in the Docker build since, by default, all the files in the same folder as the Dockerfile will be uploaded. We also discussed placing the Dockerfile in a separate folder, and the same applies to `.dockerignore`. It should go in the folder where the Dockerfile was placed.

Keeping all the items you want to use in an image in the same folder will help you keep the number of items, if any, in the `.dockerignore` file to a minimum.

Since we have spent the last few sections of this chapter looking at Dockerfiles, let's start building images using the example file we have covered here.

Using a Dockerfile

The first method that we are going to look at for building our base container images is creating a Dockerfile. In fact, we will be using the Dockerfile from the previous section and then executing a `docker image build` command against it to get ourselves an NGINX image.

So, let's start off by looking at the Dockerfile once more:

```
FROM alpine:latest
LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
LABEL description="This example Dockerfile installs NGINX."
RUN apk add --update nginx && \
    rm -rf /var/cache/apk/* && \
    mkdir -p /tmp/nginx/
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/default.conf /etc/nginx/conf.d/default.conf ADD
files/html.tar.gz /usr/share/nginx/

EXPOSE 80/tcp

ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

Don't forget that you will also need the `default.conf`, `html.tar.gz`, and `nginx.conf` files in the `files` folder. You can find these in the accompanying GitHub repository.

So, there are two ways we can go about building our image. The first way would be by specifying the `--file` switch when we use the `docker image build` command. We will also utilize the `--tag` switch to give the new image a unique name:

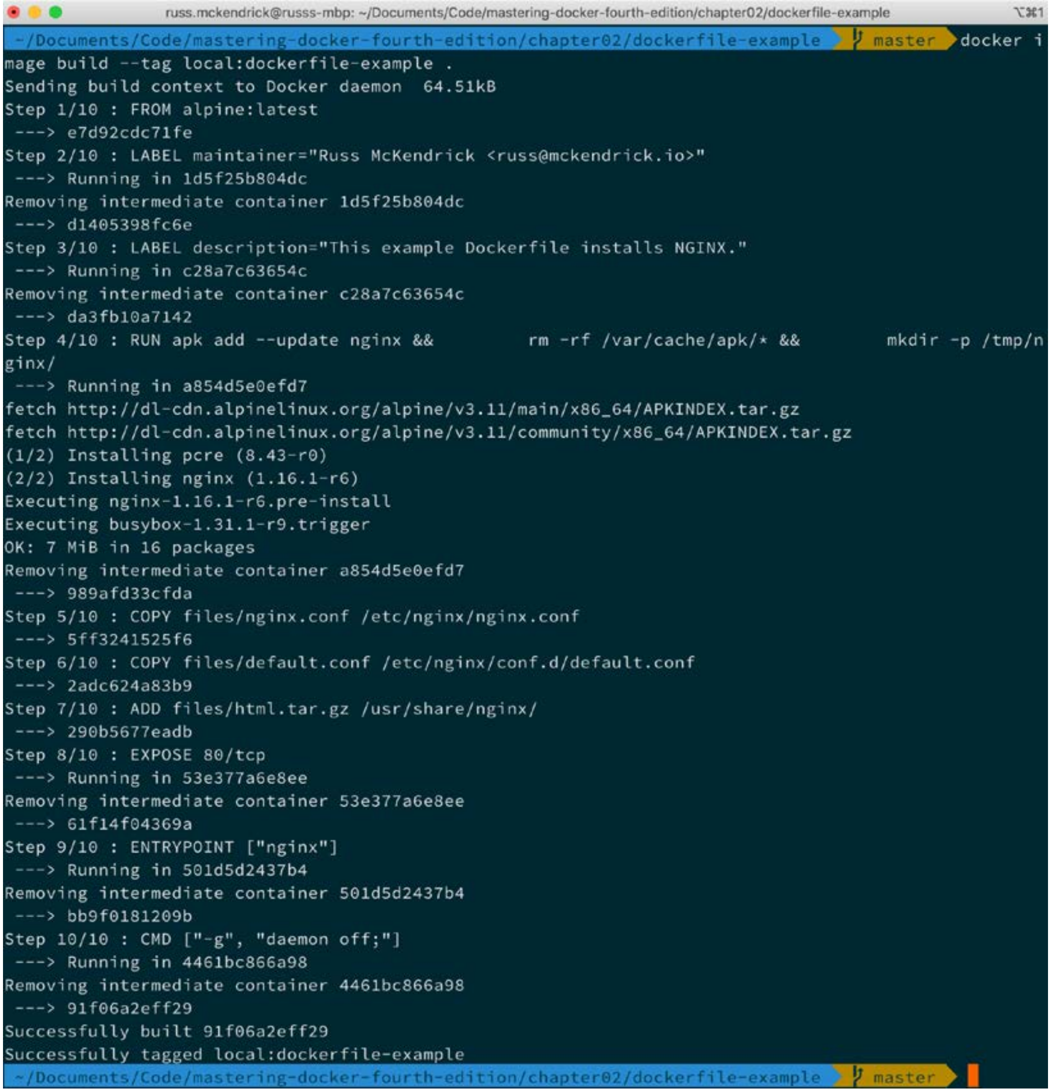
```
$ docker image build --file <path_to_Dockerfile> --tag
<REPOSITORY>:<TAG> .
```

Now, `<REPOSITORY>` is typically the username you sign up for on Docker Hub. We will look at this in more detail in *Chapter 3*, *Storing and Distributing Images*, but for now, we will be using `local`. `<TAG>` is a unique value that allows you to identify a container. Typically, this will be a version number or an other descriptor.

As we have a file called Dockerfile, we can also skip using the `--file` switch. This is the second way of building an image. The following is the code for this:

```
$ docker image build --tag local:dockerfile-example .
```

The most important thing to remember is the dot (or period) at the very end. This is to tell the `docker image build` command to build in the current folder. When you build your image, you should see something similar to the following Terminal output:



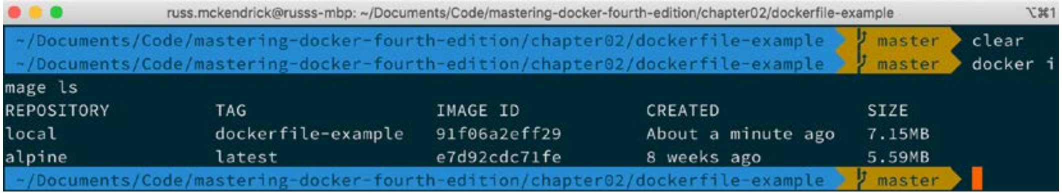
```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example   ⌥⌘1
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master docker i
mage build --tag local:dockerfile-example .
Sending build context to Docker daemon  64.51kB
Step 1/10 : FROM alpine:latest
 ---> e7d92cdc71fe
Step 2/10 : LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
 ---> Running in 1d5f25b804dc
Removing intermediate container 1d5f25b804dc
 ---> d1405398fc6e
Step 3/10 : LABEL description="This example Dockerfile installs NGINX."
 ---> Running in c28a7c63654c
Removing intermediate container c28a7c63654c
 ---> da3fb10a7142
Step 4/10 : RUN apk add --update nginx &&        rm -rf /var/cache/apk/* &&        mkdir -p /tmp/n
ginx/
 ---> Running in a854d5e0efd7
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/community/x86_64/APKINDEX.tar.gz
(1/2) Installing pcre (8.43-r0)
(2/2) Installing nginx (1.16.1-r6)
Executing nginx-1.16.1-r6.pre-install
Executing busybox-1.31.1-r9.trigger
OK: 7 MiB in 16 packages
Removing intermediate container a854d5e0efd7
 ---> 989afd33cfda
Step 5/10 : COPY files/nginx.conf /etc/nginx/nginx.conf
 ---> 5ff3241525f6
Step 6/10 : COPY files/default.conf /etc/nginx/conf.d/default.conf
 ---> 2adc624a83b9
Step 7/10 : ADD files/html.tar.gz /usr/share/nginx/
 ---> 290b5677eadb
Step 8/10 : EXPOSE 80/tcp
 ---> Running in 53e377a6e8ee
Removing intermediate container 53e377a6e8ee
 ---> 61f14f04369a
Step 9/10 : ENTRYPOINT ["nginx"]
 ---> Running in 501d5d2437b4
Removing intermediate container 501d5d2437b4
 ---> bb9f0181209b
Step 10/10 : CMD ["-g", "daemon off;"]
 ---> Running in 4461bc866a98
Removing intermediate container 4461bc866a98
 ---> 91f06a2eff29
Successfully built 91f06a2eff29
Successfully tagged local:dockerfile-example
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master
```

Figure 2.3 – Building an image from our Dockerfile

Once it's built, you should be able to run the following command to check whether the image is available, as well as the size of your image:

```
$ docker image ls
```

As you can see from the following Terminal output, my image size is 7.15 MB:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example
⌥⌘1

```
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master  clear
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master  docker i
mage ls
REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
local               dockerfile-example  91f06a2eff29        About a minute ago   7.15MB
alpine              latest              e7d92cdc71fe        8 weeks ago          5.59MB
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master
```

Figure 2.4 – Checking the size of the container image

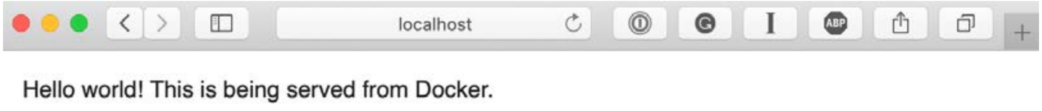
You can launch a container with your newly built image by running this command:

```
$ docker container run -d --name dockerfile-example -p 8080:80
local:dockerfile-example
```

This will launch a container called `dockerfile-example`. You can check whether it is running by using the following command:

```
$ docker container ls
```

Opening your browser and going to `http://localhost:8080/` should show you an extremely simple web page that looks as follows:



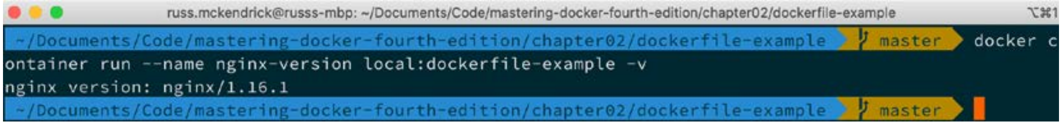
Hello world! This is being served from Docker.

Figure 2.5 – Checking the container in the browser

Next up, we will quickly run a few of the commands we covered in the *Introducing Dockerfiles* section of this chapter, starting with the following command:

```
$ docker container run --name nginx-version local:dockerfile-
example -v
```

As you can see from the following Terminal output, we are currently running NGINX version 1.16.1:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  ⌥⌘1
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master  docker c
ontainer run --name nginx-version local:dockerfile-example -v
nginx version: nginx/1.16.1
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master
```

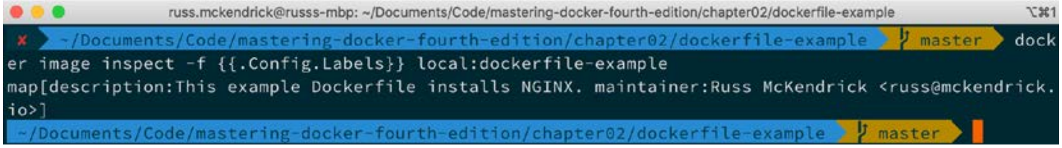
Figure 2.6 – Checking the version of NGINX

The next command we will look at running displays the labels that we embedded at build time.

To view this information, run the following command:

```
$ docker image inspect -f {{.Config.Labels}} local:dockerfile-
example
```

As you can see from the following output, this displays the information we entered:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  ⌥⌘1
 x  ~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master  dock
er image inspect -f {{.Config.Labels}} local:dockerfile-example
map[description:This example Dockerfile installs NGINX. maintainer:Russ McKendrick <russ@mckendrick.
io>]
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master
```

Figure 2.7 – Checking the labels of our newly built image

Before we move on, you can stop and remove the containers we launched with the following commands:

```
$ docker container stop dockerfile-example
$ docker container rm dockerfile-example nginx-version
```

We will go into more detail about Docker container commands in *Chapter 4*, *Managing Containers*.

Using an existing container

The easiest way to build a base image is to start off by using one of the official images from Docker Hub. Docker also keeps the Dockerfile for these official builds in their GitHub repositories. So, there are at least two choices you have for using existing images that others have already created. By using the Dockerfile, you can see exactly what is included in the build and add what you need. You can then version control that Dockerfile if you want to change or share it later.

There is another way to achieve this; however, it is not recommended or considered to be good practice, and I would strongly discourage you from using it.

> **Tip**
>
> I would only use this method during a prototyping phase to check that the commands you are running work as expected in an interactive shell before putting them in a Dockerfile. You should always use a Dockerfile.

First, we should download the image we want to use as our base; as we did previously, we will be using Alpine Linux:

```
$ docker image pull alpine:latest
```

Next, we need to run a container in the foreground so that we can interact with it:

```
$ docker container run -it --name alpine-test alpine /bin/sh
```

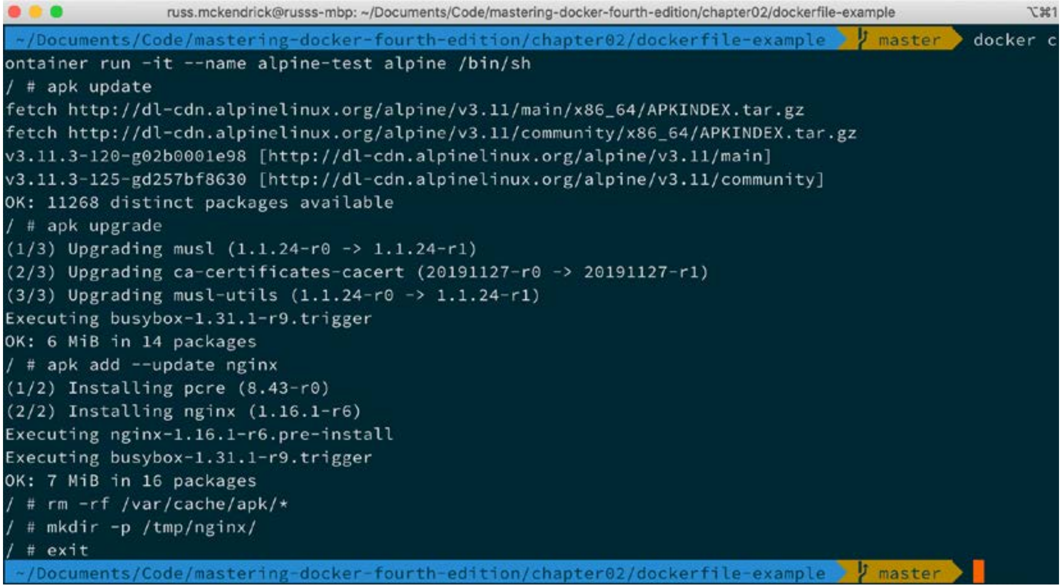
Once the container runs, you can add the packages as necessary using the apk command, or whatever the package management commands are for your Linux flavor.

For example, the following commands would install NGINX:

```
$ apk update
$ apk upgrade
$ apk add --update nginx
$ rm -rf /var/cache/apk/*
$ mkdir -p /tmp/nginx/
$ exit
```

After you have installed the packages you require, you need to save the container. The exit command at the end of the preceding set of commands will stop the running container since the shell process we are detaching ourselves from just happens to be the process keeping the container running in the foreground.

You can see this in the following Terminal output:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example ⌥⌘1
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master  docker c
ontainer run -it --name alpine-test alpine /bin/sh
/ # apk update
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.11/community/x86_64/APKINDEX.tar.gz
v3.11.3-120-g02b0001e98 [http://dl-cdn.alpinelinux.org/alpine/v3.11/main]
v3.11.3-125-gd257bf8630 [http://dl-cdn.alpinelinux.org/alpine/v3.11/community]
OK: 11268 distinct packages available
/ # apk upgrade
(1/3) Upgrading musl (1.1.24-r0 -> 1.1.24-r1)
(2/3) Upgrading ca-certificates-cacert (20191127-r0 -> 20191127-r1)
(3/3) Upgrading musl-utils (1.1.24-r0 -> 1.1.24-r1)
Executing busybox-1.31.1-r9.trigger
OK: 6 MiB in 14 packages
/ # apk add --update nginx
(1/2) Installing pcre (8.43-r0)
(2/2) Installing nginx (1.16.1-r6)
Executing nginx-1.16.1-r6.pre-install
Executing busybox-1.31.1-r9.trigger
OK: 7 MiB in 16 packages
/ # rm -rf /var/cache/apk/*
/ # mkdir -p /tmp/nginx/
/ # exit
~/Documents/Code/mastering-docker-fourth-edition/chapter02/dockerfile-example  master
```

Figure 2.8 – Checking the Terminal output

> **Tip**
>
> It is at this point that you should really stop; I do not recommend that you use the preceding commands to create and distribute images, apart from the one use case we will discuss in a moment.

So, to save our stopped container as an image, you need to do something similar to the following:

```
$ docker container commit <container_name> <REPOSITORY>:<TAG>
```

For example, I ran the following command to save a copy of the container we launched and customized:

```
$ docker container commit alpine-test local:broken-container
```

Noticed how I called my image `broken-container`? Since one of the use cases for taking this approach is that if, for some reason, you have a problem with a container, then it is extremely useful to save the failed container as an image, or even export it as a `TAR` file to share with others if you need some assistance in getting to the root of the problem.

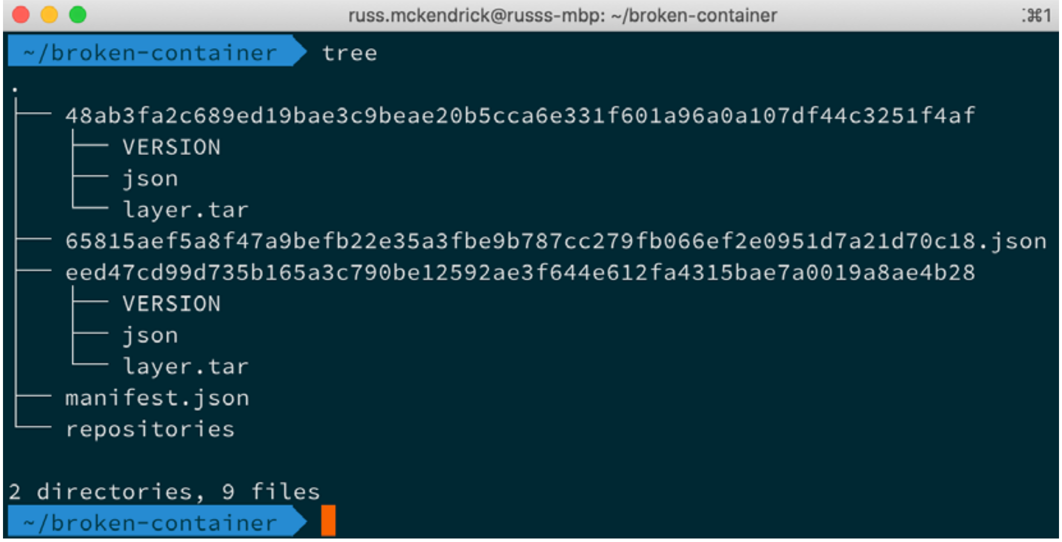
To save the image file, simply run the following command:

```
$ docker image save -o <name_of_file.tar> <REPOSITORY>:<TAG>
```

So, for our example, I ran the following command:

```
$ docker image save -o broken-container.tar local:broken-
container
```

This gave me a 7.9 MB file called `broken-container.tar`. Since we have this file, we can uncompress it and have a look around. It will have the following structure:



```
~/broken-container  tree
.
├── 48ab3fa2c689ed19bae3c9beae20b5cca6e331f601a96a0a107df44c3251f4af
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── 65815aef5a8f47a9befb22e35a3fbe9b787cc279fb066ef2e0951d7a21d70c18.json
├── eed47cd99d735b165a3c790be12592ae3f644e612fa4315bae7a0019a8ae4b28
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── manifest.json
└── repositories

2 directories, 9 files
~/broken-container
```

Figure 2.9 – Collection of JSON files, folders, and TAR files

The image is made up of a collection of JSON files, folders, and other TAR files. All the images follow this structure, so you may be thinking to yourself, **why is this method so bad?**

The biggest reason is trust (as we've already mentioned). Your end user will not be able to easily see what is in the image they are running. Would you randomly download a prepackaged image from an unknown source to run your workload, without checking how the image was built? Who knows how it was configured and what packages have been installed!

With a Dockerfile, you can see exactly what was executed to create the image, but with the method described here, you have zero visibility of this.

Another reason is that it is difficult for you to build in a good set of defaults. For example, if you were to build your image this way, then you would not really be able to take advantage of features such as `ENTRYPOINT` and `CMD`, or even the most basic instructions, such as `EXPOSE`. Instead, the user would have to define everything required while running their `docker container run` command.

In the early days of Docker, distributing images that had been prepared in this way was common practice. In fact, I was guilty of it myself since, coming from an operations background, it made perfect sense to launch a **machine**, bootstrap it, and then create a gold master. Luckily, over the last few years, Docker has extended the build functionality to the point where this option is not even considered anymore.

Using scratch as a base

So far, we have been using prepared images from Docker Hub as our base images. However, it is best to avoid this altogether (sort of) and roll out your own images from scratch.

Now, when you usually hear the phrase from scratch, it literally means that you start from nothing. That's what we have here – you get absolutely nothing and have to build upon it. Now, this can be a benefit because it will keep the image size very small, but it can also be detrimental if you are fairly new to Docker as it can get complicated.

Docker has done some of the hard work for us already and created an empty `TAR` file on Docker Hub named `scratch`; you can use it in the `FROM` section of your Dockerfile. You can base your entire Docker build on this, and then add parts as needed.

Again, we'll be using Alpine Linux as our base operating system for the image. The reasons for doing this include not only the fact that it is distributed as an ISO, Docker image, and various virtual machine images, but also that the entire operating system is available as a compressed `TAR` file. You can find the download in this book's GitHub repository, or on the Alpine Linux download page.

To download a copy, just select the appropriate download from the downloads page, which can be found at `https://www.alpinelinux.org/downloads/`. The one I used was `x86_64` from the **MINI ROOT FILESYSTEM** section.

Once it's finished downloaded, you need to create a Dockerfile that uses `scratch` and then add the `tar.gz` file, making sure to use the correct file, as shown in the following example:

```
FROM scratch
ADD files/alpine-minirootfs-3.11.3-x86_64.tar.gz /
CMD [“/bin/sh”]
```

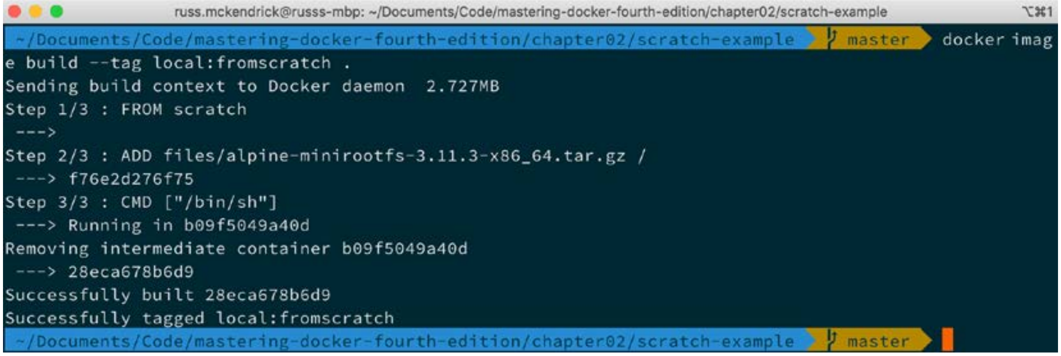
You might be thinking, why did I just download the `alpine-minirootfs-3.11.3-x86_64.tar.gz` file? Could I have not had used `http://dl-cdn.alpinelinux.org/alpine/v3.11/releases/x86_64/alpine-minirootfs-3.11.3-x86_64.tar.gz` instead?

Remember, remote archives are treated as files and are just downloaded. Normally, that wouldn't be a problem as we could just add a `RUN` command to uncompress the file, but since we are using `scratch`, an operating system hasn't been installed, which means that there are no commands available for `RUN` to be able to execute anything.

Now that we have our Dockerfile, we can build our image as we would have done on any other Docker image – by running the following command:

```
$ docker image build --tag local:fromscratch .
```

This should give you the following output:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example
⌥⌘1

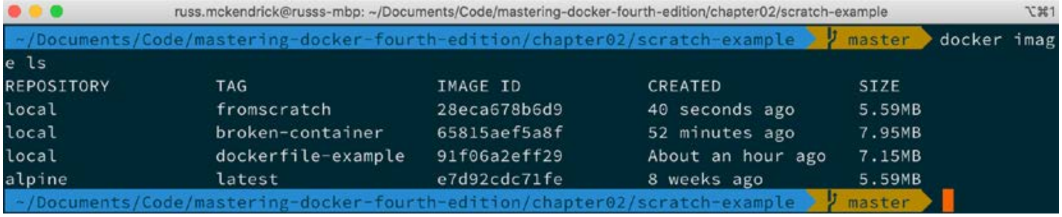
```
~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example  master  docker imag
e build --tag local:fromscratch .
Sending build context to Docker daemon  2.727MB
Step 1/3 : FROM scratch
 --->
Step 2/3 : ADD files/alpine-minirootfs-3.11.3-x86_64.tar.gz /
 ---> f76e2d276f75
Step 3/3 : CMD ["/bin/sh"]
 ---> Running in b09f5049a40d
Removing intermediate container b09f5049a40d
 ---> 28eca678b6d9
Successfully built 28eca678b6d9
Successfully tagged local:fromscratch
~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example  master
```

Figure 2.10 – Building from scratch

You can compare the image size to the other container images we have built by running the following command:

```
$ docker image ls
```

As you can see in the following screenshot, the image I built is exactly the same size as the Alpine Linux image we have been using from Docker Hub:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example
⌥⌘1

```
~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example  master  docker imag
e ls
REPOSITORY          TAG                   IMAGE ID            CREATED             SIZE
local               fromscratch           28eca678b6d9        40 seconds ago      5.59MB
local               broken-container      65815aef5a8f        52 minutes ago      7.95MB
local               dockerfile-example    91f06a2eff29        About an hour ago   7.15MB
alpine              latest                e7d92cdc71fe        8 weeks ago         5.59MB
~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example  master
```

Figure 2.11 – Reviewing the image sizes

Now that our own image has been built, we can test it by running this command:

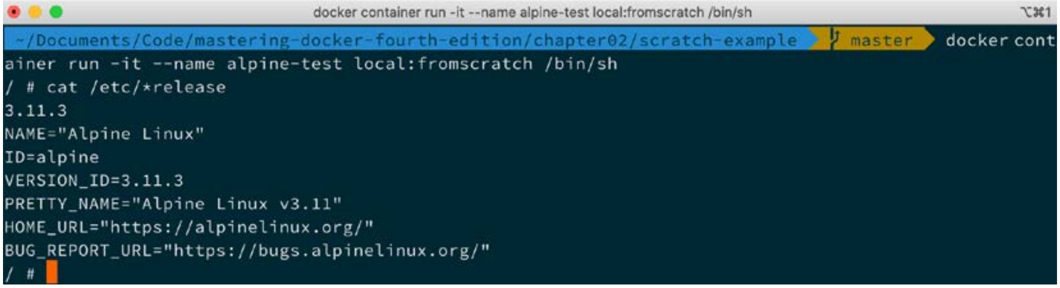
```
$ docker container run -it --name alpine-test local:fromscratch
/bin/sh
```

If you get an error, then you may already have a container called `alpine-test` created or running. Remove it by running `docker container stop alpine-test`, followed by `docker container rm alpine-test`.

This should launch us into a shell on the Alpine Linux image. You can check this by running the following command:

```
$ cat /etc/*release
```

This will display information on the release the container is running. To get an idea of what this entire process looks like, see the following Terminal output:


docker container run -it --name alpine-test local:fromscratch /bin/sh
⌥⌘1

```
~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example  master  docker cont
ainer run -it --name alpine-test local:fromscratch /bin/sh
/ # cat /etc/*release
3.11.3
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.11.3
PRETTY_NAME="Alpine Linux v3.11"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
/ #
```

Figure 2.12 – Running the image from scratch

While everything appears straightforward, this is only thanks to the way Alpine Linux packages their operating system. It can start to get more complicated when you choose to use other distributions who package their operating systems in a different way.

There are several tools that can be used to generate a bundle of an operating system. We are not going to go into any details regarding how to use any of these tools here because, if you must consider this approach, you probably have some pretty specific requirements. You can check the list of tools in the *Further reading* section at the end of this chapter for more details.

So, what could those requirements be? For most people, it will be legacy applications; for example, what happens if you have an application that requires an operating system that is no longer supported or available from Docker Hub, but you need a more modern platform to support the application? Well, you should be able to spin your image and install the application there, thus allowing you to host your old legacy application on a modern, supportable operating system/architecture.

Using ENVs

In this section, we will cover a very powerful set of variables known as **ENVs** (**ENVs**), as you will be seeing a lot of them. You can use ENVs for a lot of things in your Dockerfile. If you are familiar with coding, these will probably be familiar to you.

For others like myself, at first, they seemed intimidating, but don't get discouraged. They will become a great resource once you get the hang of them. They can be used to set information when running the container, which means that you don't have to go and update lots of the commands in your Dockerfile or in the scripts that you run on the server.

To use ENVs in your Dockerfile, you can use the `ENV` instruction. The structure of the `ENV` instruction is as follows:

```
ENV <key> <value>
ENV username admin
```

Alternatively, you can always place an equals sign between the key and the value:

```
ENV <key>=<value>
ENV username=admin
```

Now, the question is, why are there two ways that you can define them, and what are the differences?

- With the first example, you can only set one ENV per line; however, it is easy to read and follow.
- With the second ENV example, you can set multiple environmental variables on the same line, as shown here:

```
ENV username=admin database=wordpress tableprefix=wp
```

You can view which ENVs are set on an image using the `docker inspect` command:

```
$ docker image inspect <IMAGE_ID>
```

Now that we know how they need to be set in our Dockerfile, let's take a look at them in action. So far, we have been using a Dockerfile to build a simple image with just NGINX installed. Now, let's look at building something a little more dynamic.

Using Alpine Linux, we will do the following:

1. Set an ENV to define which version of PHP we would like to install.
2. Install Apache2 and our chosen PHP version.
3. Set up the image so that Apache2 starts without issues.
4. Remove the default `index.html` file and add an `index.php` file that displays the results of the `phpinfo` command.
5. Expose port 80 on the container.
6. Set Apache so that it is the default process:

> **Information**
>
> Please note that PHP5 is no longer supported. Because of that, we are having to use an older version of Alpine Linux, 3.8, as that is the last version that supports the PHP5 packages.

```
FROM alpine:3.8
LABEL maintainer=”Russ McKendrick <russ@mckendrick.io>”
LABEL description=”This example Dockerfile installs Apache &
PHP.”
ENV PHPVERSION 7

RUN apk add --update apache2 php${PHPVERSION}-apache2
php${PHPVERSION} && \
        rm -rf /var/cache/apk/* && \
        mkdir /run/apache2/ && \
        rm -rf /var/www/localhost/htdocs/index.html && \
        echo “<?php phpinfo(); ?>” > /var/www/localhost/htdocs/
index.php && \
        chmod 755 /var/www/localhost/htdocs/index.php

EXPOSE 80/tcp

ENTRYPOINT [“httpd”]
CMD [“-D”, “FOREGROUND”]
```

As you can see, we have chosen to install PHP7; we can build the image by running the following command:

```
$ docker build --tag local/apache-php:7 .
```

Notice how we have changed the command slightly. This time, we are calling the `local/apache-php` image and tagging the version as `7`. The full output that we obtained by running the preceding command is as follows:

```
Sending build context to Docker daemon   2.56kB
Step 1/8 : FROM alpine:3.8
 ---> c8bccc0af957
Step 2/8 : LABEL maintainer=”Russ McKendrick <russ@mckendrick.
io>”
 ---> Running in 7746dd8cabd0
Removing intermediate container 7746dd8cabd0
 ---> 9173a415ed21
Step 3/8 : LABEL description=”This example Dockerfile installs
Apache & PHP.”
 ---> Running in 322e98b9c2e0
Removing intermediate container 322e98b9c2e0
 ---> aefb9450e664
Step 4/8 : ENV PHPVERSION 7
```

As you can see from the preceding output, the `PHPVERSION` ENV has been set to the number `7`:

```
 ---> Running in 0b9e9a5d8956
Removing intermediate container 0b9e9a5d8956
 ---> 219afdf8eeb8
Step 5/8 : RUN apk add --update apache2 php${PHPVERSION}-
apache2 php${PHPVERSION} &&         rm -rf /var/cache/apk/* &&
         mkdir /run/apache2/ &&         rm -rf /var/www/
localhost/htdocs/index.html &&         echo “<?php phpinfo();
?>” > /var/www/localhost/htdocs/index.php &&         chmod 755
/var/www/localhost/htdocs/index.php
 ---> Running in 36823df46b29
fetch http://dl-cdn.alpinelinux.org/alpine/v3.8/main/x86_64/
APKINDEX.tar.gz
```

```
fetch http://dl-cdn.alpinelinux.org/alpine/v3.8/community/
x86_64/APKINDEX.tar.gz
(1/14) Installing libuuid (2.32-r0)
(2/14) Installing apr (1.6.3-r1)
(3/14) Installing expat (2.2.8-r0)
(4/14) Installing apr-util (1.6.1-r3)
(5/14) Installing pcre (8.42-r0)
(6/14) Installing apache2 (2.4.41-r0)
Executing apache2-2.4.41-r0.pre-install
```

So far, we have only referred to the ENV. As shown by the following output, the necessary `php7` packages will start to be installed:

```
(7/14) Installing php7-common (7.2.26-r0)
(8/14) Installing ncurses-terminfo-base (6.1_p20180818-r1)
(9/14) Installing ncurses-terminfo (6.1_p20180818-r1)
(10/14) Installing ncurses-libs (6.1_p20180818-r1)
(11/14) Installing libedit (20170329.3.1-r3)
(12/14) Installing libxml2 (2.9.8-r2)
(13/14) Installing php7 (7.2.26-r0)
(14/14) Installing php7-apache2 (7.2.26-r0)
Executing busybox-1.28.4-r3.trigger
OK: 26 MiB in 27 packages
Removing intermediate container 36823df46b29
```

Now that all the packages have been installed, the build can do some housekeeping and then complete:

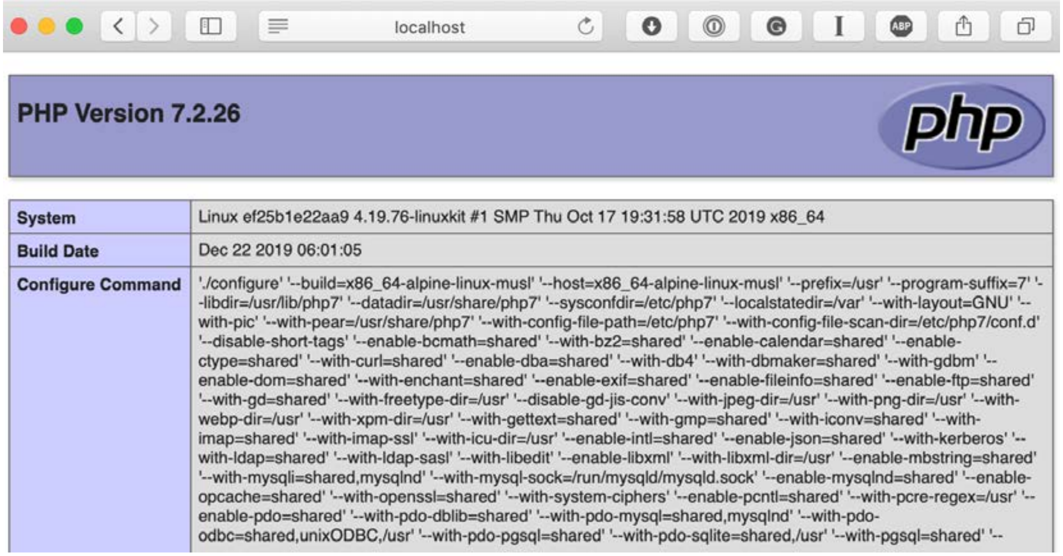
```
 ---> 842eebf1d363
Step 6/8 : EXPOSE 80/tcp
 ---> Running in 40494d3b357f
Removing intermediate container 40494d3b357f
 ---> 074e10ff8526
Step 7/8 : ENTRYPOINT [“httpd”]
 ---> Running in a56700cae985
Removing intermediate container a56700cae985
 ---> 25b63b51f243
Step 8/8 : CMD [“-D”, “FOREGROUND”]
```

```
 ---> Running in d2c478e67c0c
Removing intermediate container d2c478e67c0c
 ---> 966dcf5cafdf
Successfully built 966dcf5cafdf
Successfully tagged local/apache-php:7
```

We can check whether everything ran as expected by running the following command to launch a container using the image:

```
$ docker container run -d -p 8080:80 --name apache-php7 local/
apache-php:7
```

Once it's launched, open a browser and go to `http://localhost:8080/`. You should see a page showing that PHP7 is being used:



PHP Version 7.2.26

System	Linux ef25b1e22aa9 4.19.76-linuxkit #1 SMP Thu Oct 17 19:31:58 UTC 2019 x86_64
Build Date	Dec 22 2019 06:01:05
Configure Command	'./configure' '--build=x86_64-alpine-linux-musl' '--host=x86_64-alpine-linux-musl' '--prefix=/usr' '--program-suffix=7' '--libdir=/usr/lib/php7' '--datadir=/usr/share/php7' '--sysconfdir=/etc/php7' '--localstatedir=/var' '--with-layout=GNU' '--with-pic' '--with-pear=/usr/share/php7' '--with-config-file-path=/etc/php7' '--with-config-file-scan-dir=/etc/php7/conf.d' '--disable-short-tags' '--enable-bcmath=shared' '--with-bz2=shared' '--enable-calendar=shared' '--enable-ctype=shared' '--with-curl=shared' '--enable-dba=shared' '--with-db4' '--with-dbmaker=shared' '--with-gdbm' '--enable-dom=shared' '--with-enchant=shared' '--enable-exif=shared' '--enable-fileinfo=shared' '--enable-ftp=shared' '--with-gd=shared' '--with-freetype-dir=/usr' '--disable-gd-jis-conv' '--with-jpeg-dir=/usr' '--with-png-dir=/usr' '--with-webp-dir=/usr' '--with-xpm-dir=/usr' '--with-gettext=shared' '--with-gmp=shared' '--with-iconv=shared' '--with-imap=shared' '--with-imap-ssl' '--with-icu-dir=/usr' '--enable-intl=shared' '--enable-json=shared' '--with-kerberos' '--with-ldap=shared' '--with-ldap-sasl' '--with-libedit' '--enable-libxml' '--with-libxml-dir=/usr' '--enable-mbstring=shared' '--with-mysqli=shared,mysqlnd' '--with-mysql-sock=/run/mysqld/mysqld.sock' '--enable-mysqlnd=shared' '--enable-opcache=shared' '--with-openssl=shared' '--with-system-ciphers' '--enable-pcntl=shared' '--with-pcre-regex=/usr' '--enable-pdo=shared' '--with-pdo-dblib=shared' '--with-pdo-mysql=shared,mysqlnd' '--with-pdo-odbc=shared,unixODBC,/usr' '--with-pdo-pgsql=shared' '--with-pdo-sqlite=shared,/usr' '--with-pgsql=shared' '--

Figure 2.13 – Checking the PHP version

Tip

Don't be confused by the next part; there is no PHP6. You can find out more about this at the following RFC and the results of the vote for skipping PHP6 at `https://wiki.php.net/rfc/php6`.

Now, in your Dockerfile, change `PHPVERSION` from `7` to `5` and then run the following command to build a new image:

```
$ docker image build --tag local/apache-php:5 .
```

As you can see from the following Terminal output, the majority of the output is the same, apart from the packages that are being installed:

```
Sending build context to Docker daemon   2.56kB
Step 1/8 : FROM alpine:3.8
 ---> c8bccc0af957
Step 2/8 : LABEL maintainer=”Russ McKendrick <russ@mckendrick.
io>”
 ---> Using cache
 ---> 9173a415ed21
Step 3/8 : LABEL description=”This example Dockerfile installs
Apache & PHP.”
 ---> Using cache
 ---> aefb9450e664
Step 4/8 : ENV PHPVERSION 5
```

Here, we can see that `5` has been set as the value of the `PHPVERSION` ENV. From here, the build will continue just like the previous build did:

```
 ---> Running in d6e8dc8b70ce
Removing intermediate container d6e8dc8b70ce
 ---> 71896c898e35
Step 5/8 : RUN apk add --update apache2 php${PHPVERSION}-
apache2 php${PHPVERSION} &&         rm -rf /var/cache/apk/* &&
         mkdir /run/apache2/ &&         rm -rf /var/www/
localhost/htdocs/index.html &&         echo “<?php phpinfo();
 ?>” > /var/www/localhost/htdocs/index.php &&         chmod 755
 /var/www/localhost/htdocs/index.php
 ---> Running in fb946c0684e4
fetch http://dl-cdn.alpinelinux.org/alpine/v3.8/main/x86_64/
APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.8/community/
x86_64/APKINDEX.tar.gz
(1/15) Installing libuuid (2.32-r0)
```

```
(2/15) Installing apr (1.6.3-r1)
(3/15) Installing expat (2.2.8-r0)
(4/15) Installing apr-util (1.6.1-r3)
(5/15) Installing pcre (8.42-r0)
(6/15) Installing apache2 (2.4.41-r0)
Executing apache2-2.4.41-r0.pre-install
```

Here is where the PHP5 packages are installed. This is the only difference between our two builds:

```
(7/15) Installing php5-common (5.6.40-r0)
(8/15) Installing ncurses-terminfo-base (6.1_p20180818-r1)
(9/15) Installing ncurses-terminfo (6.1_p20180818-r1)
(10/15) Installing ncurses-libs (6.1_p20180818-r1)
(11/15) Installing readline (7.0.003-r0)
(12/15) Installing libxml2 (2.9.8-r2)
(13/15) Installing php5-cli (5.6.40-r0)
(14/15) Installing php5 (5.6.40-r0)
(15/15) Installing php5-apache2 (5.6.40-r0)
Executing busybox-1.28.4-r3.trigger
OK: 48 MiB in 28 packages
Removing intermediate container fb946c0684e4

```

Again, now that the packages have been installed, the build will progress as it did previously until we have our complete image:

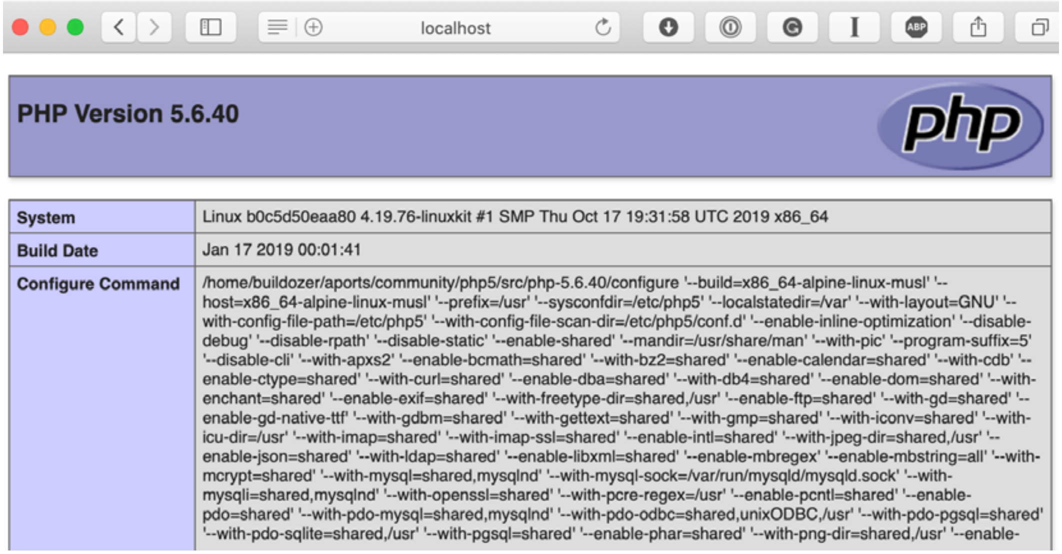
```
 ---> 54cbb6ef4724
Step 6/8 : EXPOSE 80/tcp
 ---> Running in 59776669f08a
Removing intermediate container 59776669f08a
 ---> e34c5c34658d
Step 7/8 : ENTRYPOINT [“httpd”]
 ---> Running in 037ecfed197c
Removing intermediate container 037ecfed197c
 ---> c50bdf3e4b02
Step 8/8 : CMD [“-D”, “FOREGROUND”]
 ---> Running in 9eccc9131ef9
```

```
Removing intermediate container 9eccc9131ef9
 ---> 7471b75e789e
Successfully built 7471b75e789e
Successfully tagged local/apache-php:5
```

We can launch a container, this time on port 9090, by running the following command:

```
$ docker container run -d -p 9090:80 --name apache-php5 local/
apache-php:5
```

Opening your browser again, but this time going to `http://localhost:9090/`, should show that we are running PHP5:



PHP Version 5.6.40

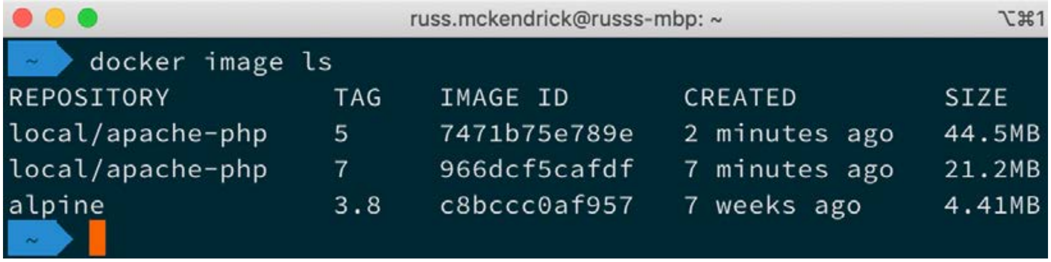
System	Linux b0c5d50eaa80 4.19.76-linuxkit #1 SMP Thu Oct 17 19:31:58 UTC 2019 x86_64
Build Date	Jan 17 2019 00:01:41
Configure Command	/home/buildozer/aports/community/php5/src/php-5.6.40/configure '--build=x86_64-alpine-linux-musl' '--host=x86_64-alpine-linux-musl' '--prefix=/usr' '--sysconfdir=/etc/php5' '--localstatedir=/var' '--with-layout=GNU' '--with-config-file-path=/etc/php5' '--with-config-file-scan-dir=/etc/php5/conf.d' '--enable-inline-optimization' '--disable-debug' '--disable-rpath' '--disable-static' '--enable-shared' '--mandir=/usr/share/man' '--with-pic' '--program-suffix=5' '--disable-cli' '--with-apxs2' '--enable-bcmath=shared' '--with-bz2=shared' '--enable-calendar=shared' '--with-cdb' '--enable-ctype=shared' '--with-curl=shared' '--enable-dba=shared' '--with-db4=shared' '--enable-dom=shared' '--with-enchant=shared' '--enable-exif=shared' '--with-freetype-dir=shared,/usr' '--enable-ftp=shared' '--with-gd=shared' '--enable-gd-native-ttf' '--with-gdbm=shared' '--with-gettext=shared' '--with-gmp=shared' '--with-iconv=shared' '--with-icu-dir=/usr' '--with-imap=shared' '--with-imap-ssl=shared' '--enable-intl=shared' '--with-jpeg-dir=shared,/usr' '--enable-json=shared' '--with-ldap=shared' '--enable-libxml=shared' '--enable-mbregex' '--enable-mbstring=all' '--with-mcrypt=shared' '--with-mysql=shared,mysqlnd' '--with-mysql-sock=/var/run/mysqld/mysqld.sock' '--with-mysqli=shared,mysqlnd' '--with-openssl=shared' '--with-pcre-regex=/usr' '--enable-pcntl=shared' '--enable-pdo=shared' '--with-pdo-mysql=shared,mysqlnd' '--with-pdo-odbc=shared,unixODBC,/usr' '--with-pdo-pgsql=shared' '--with-pdo-sqlite=shared,/usr' '--with-pgsql=shared' '--enable-phar=shared' '--with-png-dir=shared,/usr' '--enable-

Figure 2.14 – Running PHP5

Finally, you can compare the size of the images by running this command:

```
$ docker image ls
```

You should see the following Terminal output:



```
~ docker image ls
REPOSITORY          TAG    IMAGE ID       CREATED          SIZE
local/apache-php    5      7471b75e789e   2 minutes ago    44.5MB
local/apache-php    7      966dcf5cafdf   7 minutes ago    21.2MB
alpine              3.8    c8bccc0af957   7 weeks ago      4.41MB
~
```

Figure 2.15 – Comparing the image sizes

This shows that the PHP7 image is a lot smaller than the PHP5 one. Let's discuss what actually happened when we built the two different container images.

So, what happened? Well, when Docker launched the Alpine Linux image to create our image, the first thing it did was set the ENVs we defined, making them available to all the shells within the container.

Luckily for us, the naming scheme for PHP in Alpine Linux simply substitutes the version number and maintains the same name for the packages we need to install, meaning that we run the following command:

```
RUN apk add --update apache2 php${PHPVERSION}-apache2
php${PHPVERSION}
```

But it is actually interpreted as follows:

```
RUN apk add --update apache2 php7-apache2 php7
```

For PHP5, it is interpreted as the following instead:

```
RUN apk add --update apache2 php5-apache2 php5
```

This means that we do not have to go through the whole Dockerfile, manually substituting version numbers. This approach is especially useful when installing packages from remote URLs, such as software release pages.

What follows is a more advanced example: a Dockerfile that installs and configures Consul by *HashiCorp*. In this Dockerfile, we are using ENVs to define the version numbers and the SHA256 hash of the file we downloaded:

```
FROM alpine:latest
LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
LABEL description="An image with the latest version on Consul."

```

```
ENV CONSUL_VERSION 1.7.1
ENV CONSUL_SHA256 09f3583c6cd7b1f748c0c012ce9b3d96de95
a6c0d2334327b74f7d72b1fa5054

RUN  apk add --update ca-certificates wget && \
     wget -O consul.zip https://releases.hashicorp.com/consul/
${CONSUL_VERSION}/consul_${CONSUL_VERSION}_linux_amd64.zip && \
     echo “$CONSUL_SHA256 *consul.zip” | sha256sum -c - && \
     unzip consul.zip && \
     mv consul /bin/ && \
     rm -rf consul.zip && \
     rm -rf /tmp/* /var/cache/apk/*

EXPOSE 8300 8301 8301/udp 8302 8302/udp 8400 8500 8600 8600/udp

VOLUME [ “/data” ]

ENTRYPOINT [ “/bin/consul” ]
CMD [ “agent”, “-data-dir”, “/data”, “-server”, “
-bootstrap-expect”, “1”, “-client=0.0.0.0”]
```

As you can see, Dockerfiles can get quite complex, and using ENVs can help with maintenance. Whenever a new version of Consul is released, I simply need to update the `ENV` line and commit it to GitHub, which will trigger a new image being built. Well – it would have done if we had configured it to do so. We will look at this in the next chapter.

You might have also noticed we are using an instruction within the Dockerfile that we have not covered here. Don't worry – we will look at the `VOLUME` instruction in *Chapter 4*, *Managing Containers*.

Using multi-stage builds

In this section, which is the final part of our journey into using Dockerfiles and building container images, we will look at using a relatively new method for building an image. In the previous sections, we looked at adding binaries directly to our images either via a package manager, such as Alpine Linux's APK, or, in the previous example, by downloading a precompiled binary from the software vendor.

What if we wanted to compile our own software as part of the build? Historically, we would have had to use a container image containing a full build environment, which can be very big. This means that we probably would have had to cobble together a script that ran through something like the following process:

1. Download the build environment container image and start a `build` container.
2. Copy the source code to the `build` container.
3. Compile the source code on the `build` container.
4. Copy the compiled binary outside of the `build` container.
5. Remove the `build` container.
6. Use a pre-written Dockerfile to build an image and copy the binary to it.

That is a lot of logic – in an ideal world, it should be part of Docker. Luckily, the Docker community thought so, and the functionality to achieve this, called a multi-stage build, was introduced in Docker 17.05.

The Dockerfile contains two different build stages:

- The first, named `builder`, uses the official Go container image from Docker Hub. Here, we are installing a prerequisite, downloading the source code directly from GitHub, and then compiling it into a static binary:

```
FROM golang:latest as builder
WORKDIR /go-http-hello-world/
RUN go get -d -v golang.org/x/net/html
ADD https://raw.githubusercontent.com/geetarista/go-http-hello-world/master/hello_world/hello_world.go ./hello_world.go
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o
app .

FROM scratch
COPY --from=builder /go-http-hello-world/app .
CMD [“./app”]
```

Tip

Notice here we are using ADD with a URL since we want to download an uncompressed version of the source code rather than a zipped archive.

As our static binary has a built-in web server, we do not really need anything else to be present from an operating system point of view. Because of this, we are able to use `scratch` as the base image, meaning that all our image will contain is the static binary that we have copied from the builder image, and won't contain any of the `builder` environment at all.

To build the image, we just need to run the following command:

```
$ docker image build --tag local:go-hello-world .
```

The output of the preceding command can be found in the following code block. The interesting bits happen between *steps 5* and *6*:

```
Sending build context to Docker daemon  2.048kB
Step 1/8 : FROM golang:latest as builder
latest: Pulling from library/golang
50e431f79093: Pull complete
dd8c6d374ea5: Pull complete
c85513200d84: Pull complete
55769680e827: Pull complete
15357f5e50c4: Pull complete
9edb2e455d9d: Pull complete
ed2acfe844ed: Pull complete
Digest: sha256:d27017d27f9c9a58b361aa36126a29587ffd3b1b274af0
d583fe4954365a4a59
Status: Downloaded newer image for golang:latest
 ---> 25c4671a1478
```

Now that the build environment container image has been pulled, we can prepare the environment to build our code:

```

Step 2/8 : WORKDIR /go-http-hello-world/
 ---> Running in 9c23e012e016
Removing intermediate container 9c23e012e016
 ---> ea0d7e26799e
Step 3/8 : RUN go get -d -v golang.org/x/net/html
 ---> Running in 17f173992763
get “golang.org/x/net/html”: found meta tag get.metaImport
{Prefix:”golang.org/x/net”, VCS:”git”,
 RepoRoot:”https://go.googlesource.com/net”} at
//golang.org/x/net/html?go-get=1
get “golang.org/x/net/html”: verifying non-authoritative meta
tag
golang.org/x/net (download)
Removing intermediate container 17f173992763
```

With the environment prepared, we can download the source code from GitHub and compile it:

```
 ---> 68f07e01b0cf
Step 4/8 : ADD https://raw.githubusercontent.com/geetarista/
go-http-hello-world/master/hello_world/hello_world.go ./hello_
world.go
Downloading     393B
 ---> 4fb92adacdb0
Step 5/8 : RUN CGO_ENABLED=0 GOOS=linux go build -a
-installsuffix cgo -o app .
 ---> Running in 61a82b417f60
Removing intermediate container 61a82b417f60
 ---> 502d219e6869
```

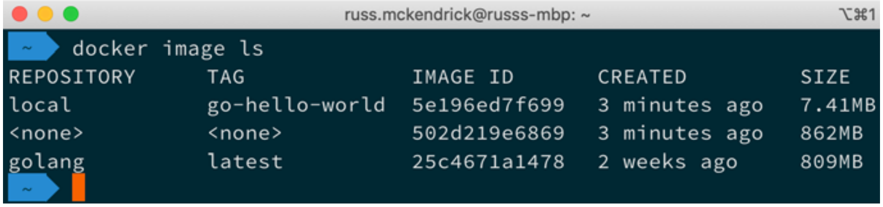
We now have our compiled code as a single executable binary, which means that we can create a new build image using `scratch` and copy the binary from the previous build image across to the new build image:

```
Step 6/8 : FROM scratch
 --->
Step 7/8 : COPY --from=builder /go-http-hello-world/app .
 ---> 2b4a6e6066e5
Step 8/8 : CMD [“./app”]
 ---> Running in c82089ea8a6b
Removing intermediate container c82089ea8a6b
 ---> 5e196ed7f699
Successfully built 5e196ed7f699
Successfully tagged local:go-hello-world
```

As you can see, our binary has been compiled and the container that contains the build environment has been removed, leaving us with an image storing our binary. If you were to run the following command, you would get an idea of why it is a good idea not to ship an application with its build environment intact:

```
$ docker image ls
```

The following output shows that the `golang` image is `809MB`; with our source code and prerequisites added, the size increases to `862MB`:



```
russ.mckendrick@russs-mbp: ~
~ docker image ls
REPOSITORY      TAG             IMAGE ID       CREATED          SIZE
local           go-hello-world  5e196ed7f699   3 minutes ago    7.41MB
<none>          <none>          502d219e6869   3 minutes ago    862MB
golang          latest          25c4671a1478   2 weeks ago      809MB
~
```

Figure 2.16 – Checking the image sizes

However, the final image is just `7.41MB`. I am sure you will agree that this is quite a dramatic amount of space that's been saved. It also adheres to the best practices by only having content relevant to our application shipped within the image, as well as being really, really small.

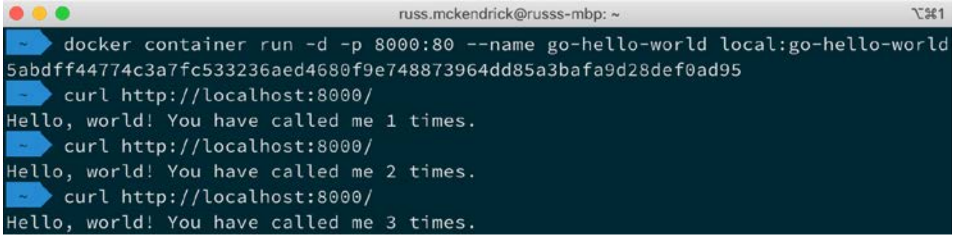
You can test the application by launching a container with the following command:

```
$ docker container run -d -p 8000:80 --name go-hello-world
local:go-hello-world
```

The application is accessible over a browser and simply increments a counter each time the page is loaded. To test it on macOS and Linux, you can use the `curl` command, as follows:

```
$ curl http://localhost:8000/
```

This should give you something like the following:



```
russ.mckendrick@russs-mbp: ~
~ docker container run -d -p 8000:80 --name go-hello-world local:go-hello-world
5abdff44774c3a7fc533236aed4680f9e748873964dd85a3bafa9d28def0ad95
~ curl http://localhost:8000/
Hello, world! You have called me 1 times.
~ curl http://localhost:8000/
Hello, world! You have called me 2 times.
~ curl http://localhost:8000/
Hello, world! You have called me 3 times.
```

Figure 2.17 – Running the container and calling the page using curl

Windows users can simply visit `http://localhost:8000/` in a browser. To stop and remove the running container, use the following commands:

```
$ docker container stop go-hello-world
$ docker container rm go-hello-world
```

As you can see, using a multi-stage build is a relatively simple process and is in keeping with the instructions that should already be starting to feel familiar to you.

Summary

In this chapter, we looked at Dockerfiles, which I am sure you will agree are a straightforward way of defining you own Docker images.

Once we finished our in-depth look at Dockerfiles, we then looked at five ways in which we can build your images. We started by looking at **using a Dockerfile** as this is the most common way you will be building your images and we will be using it throughout the rest of this book.

Then we discussed **using an existing container** as when Docker first came on the scene, this was the way most people originally built their images. It is no longer considered a best practice and should only ever be used if you need to create a snapshot of a running or crashed container for debug purposes.

Next up we talked about **using scratch as a base.** This is probably the most streamlined way of creating an image as you are literally starting from zero.

We moved onto discussing **using environmental variables**. Here, we looked at the ways we can start to introduce variables such as version numbers into our Dockerfile in a way that means we don't have to update the file in several places.

Finally, we covered **using multi-stage builds.** We used a few of the techniques we previously covered to compile an application and then copy the compiled code to a `scratch` container, giving us the smallest usable images possible.

In the next chapter, now that we know how to build images using Dockerfiles, we will be taking a look at Docker Hub and all of the advantages that using a registry service brings.

We will also look at the Docker registry, which is open source, so that you can create and configure your own place to store images, as well as third-party hosted registry services, all of which can be used to distribute you own container images.

Questions

1. True or false: the LABEL instruction tags your image once it has been built.
2. What's the difference between the ENTRYPOINT and CMD instructions?
3. True or false: when using the ADD instruction, you can't download and automatically uncompress an externally hosted archive.
4. What is a valid use for using an existing container as the base of your image?
5. What does the EXPOSE instruction expose?

Further reading

- You can find the guidelines for the official Docker container images at https://github.com/docker-library/official-images/.
- Some of the tools to help you create containers from existing installations are as follows:

  debootstrap: https://wiki.debian.org/Debootstrap

  yumbootstrap: http://dozzie.jarowit.net/trac/wiki/yumbootstrap

  rinse: https://packages.debian.org/sid/admin/rinse

  Docker contrib scripts: https://github.com/moby/moby/tree/master/contrib
- The full GitHub repository for the Go HTTP Hello World application can be found at https://github.com/geetarista/go-http-hello-world.

3
Storing and Distributing Images

In this chapter, we will cover several services, such as Docker Hub, which allows you to store your images, and Docker Registry, which you can use to run your local storage for Docker containers. We will review the differences between these services, as well as when and how to use each of them. This chapter will also cover how to set up automated builds using web hooks, as well as all the pieces that are required to set them up.

Let's take a quick look at the topics we will cover in this chapter:

- Understanding Docker Hub
- Deploying your own Docker registry
- Reviewing third-party registries
- Looking at Microbadger

Let's get started!

Technical requirements

In this chapter, we will be using our Docker installation to build images. As mentioned in the previous chapter, although the screenshots in this chapter will be from my preferred operating system, which is macOS, the commands we will be running will work on all three operating systems covered in the previous chapters.

Check out the following video to see the Code in Action:
`https://bit.ly/3iaKo9I`

Understanding Docker Hub

Although we were introduced to Docker Hub in the previous two chapters, we haven't interacted with it much other than when using the `docker image pull` command to download remote images.

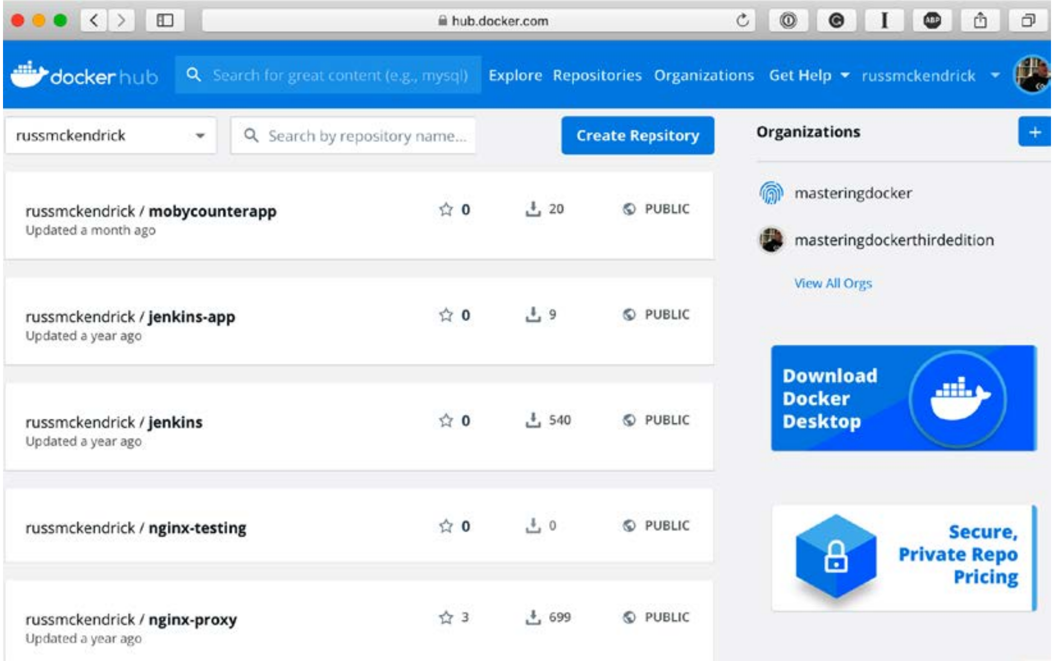
In this section, we will focus on Docker Hub, which has both a freely available option, where you can only host publicly accessible images, and also a subscription option, which allows you to host your own private images. We will focus on the web aspect of Docker Hub and the management you can do there.

The home page, which can be found at `https://hub.docker.com`, contains a **Sign-Up** form and, at the top-right, an option to **Sign in**. The odds are that if you have been dabbling with Docker, then you already have a Docker ID. If you don't, use the **Sign-Up** form on the home page to create one. If you already have a Docker ID, then simply click **Sign in**.

Once logged in, you will be presented with the main Dashboard.

The Docker Hub Dashboard

After logging in to Docker Hub, you will be taken to the following landing page. This page is known as the Dashboard of Docker Hub:



russmckendrick / **mobycounterapp** Updated a month ago	☆ 0	20	PUBLIC
russmckendrick / **jenkins-app** Updated a year ago	☆ 0	9	PUBLIC
russmckendrick / **jenkins** Updated a year ago	☆ 0	540	PUBLIC
russmckendrick / **nginx-testing**	☆ 0	0	PUBLIC
russmckendrick / **nginx-proxy** Updated a year ago	☆ 3	699	PUBLIC

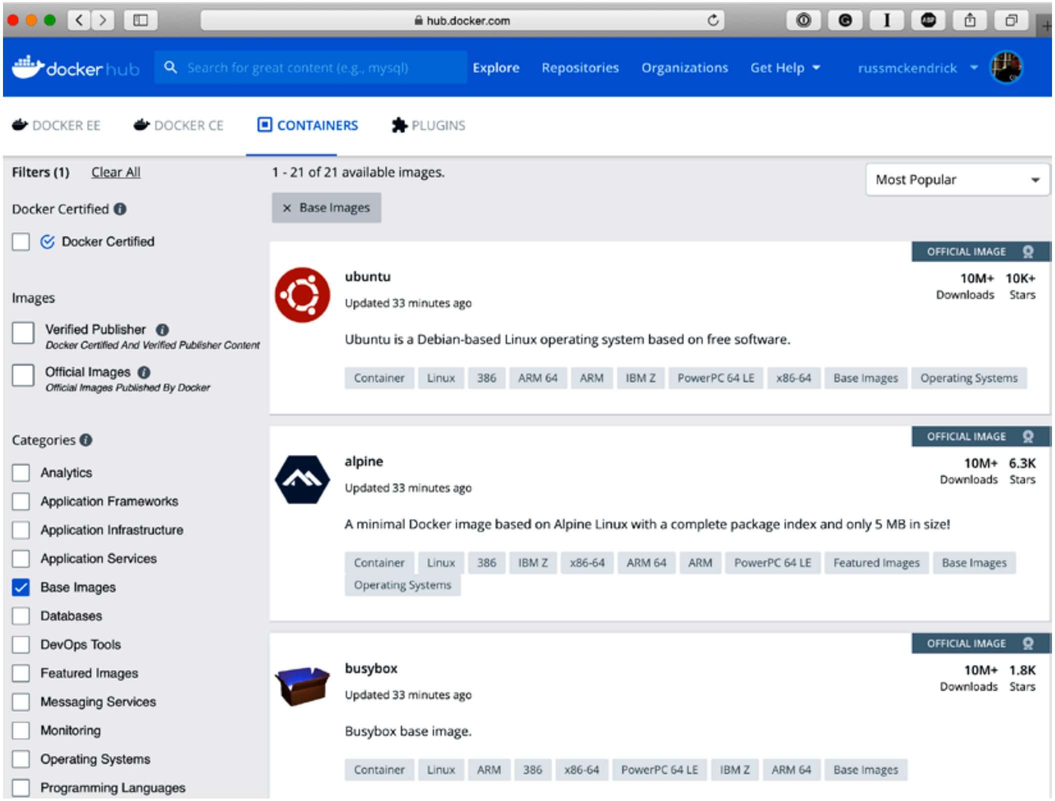
Figure 3.1 – The initial Docker Hub Dashboard

From here, you can get to all the other subpages of Docker Hub. However, before we look at those sections, we should talk a little about the Dashboard. From here, you can view all your images, both public and private. They are ordered first by the number of stars and then by the number of pulls; this order cannot be changed.

In the upcoming sections, we will go through everything you can see on the Dashboard, starting with the light blue menu at the top of the page.

Explore

The Explore option takes you to a list of official Docker images; like your Dashboard, they are ordered by stars and then pulls. As shown in the following screenshot, I have selected the base images. Each of the most popular official images has had over 10 million downloads and have thousands of stars:



hub.docker.com

docker hub | Search for great content (e.g., mysql) | Explore | Repositories | Organizations | Get Help | russmckendrick

DOCKER EE | DOCKER CE | CONTAINERS | PLUGINS

Filters (1) Clear All

Docker Certified

□ Docker Certified

Images

□ Verified Publisher
Docker Certified And Verified Publisher Content

□ Official Images
Official Images Published By Docker

Categories

□ Analytics
□ Application Frameworks
□ Application Infrastructure
□ Application Services
☑ Base Images
□ Databases
□ DevOps Tools
□ Featured Images
□ Messaging Services
□ Monitoring
□ Operating Systems
□ Programming Languages

1 - 21 of 21 available images.

Most Popular

× Base Images

OFFICIAL IMAGE
ubuntu
Updated 33 minutes ago
10M+ Downloads 10K+ Stars
Ubuntu is a Debian-based Linux operating system based on free software.
Container | Linux | 386 | ARM 64 | ARM | IBM Z | PowerPC 64 LE | x86-64 | Base Images | Operating Systems

OFFICIAL IMAGE
alpine
Updated 33 minutes ago
10M+ Downloads 6.3K Stars
A minimal Docker image based on Alpine Linux with a complete package index and only 5 MB in size!
Container | Linux | 386 | IBM Z | x86-64 | ARM 64 | ARM | PowerPC 64 LE | Featured Images | Base Images | Operating Systems

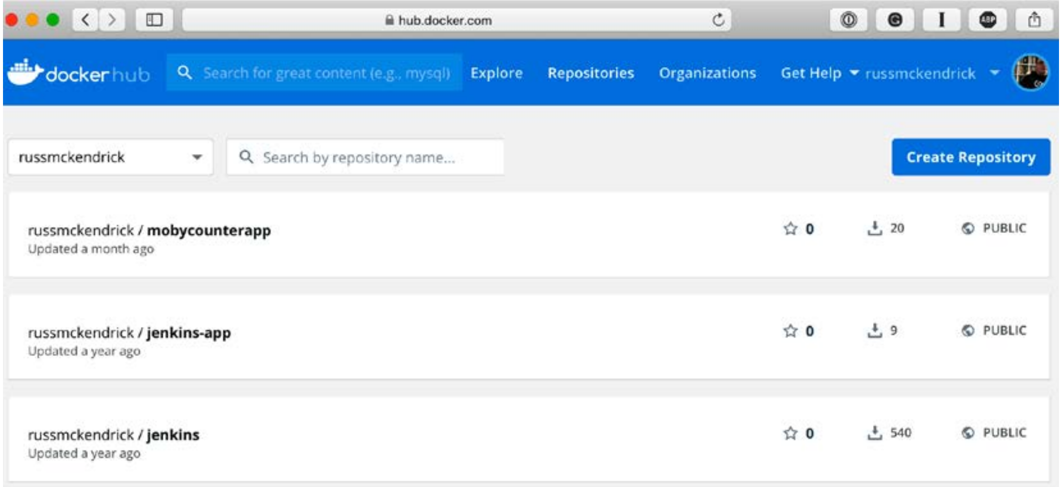
OFFICIAL IMAGE
busybox
Updated 33 minutes ago
10M+ Downloads 1.8K Stars
Busybox base image.
Container | Linux | ARM | 386 | x86-64 | PowerPC 64 LE | IBM Z | ARM 64 | Base Images

Figure 3.2 – Exploring Docker Hub

Docker Hub now integrates the Docker Store, giving you a one-stop shop for everything Docker-related from a single location rather than several different sources.

Repositories

We will go into more detail about creating a repository when in the *Creating an automated build* section of this chapter, so I will not go into any details here. This section is where you can manage your own repositories. As shown in the following screenshot, you get a quick overview of how many people have started you repository, as well as how many pulls your image has had, along with details of whether the repository is public or private:



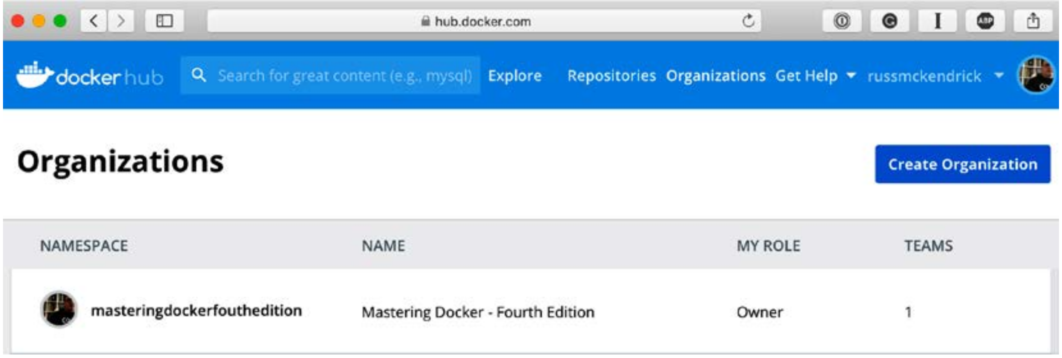
russmckendrick / **mobycounterapp** Updated a month ago	☆ 0	20	PUBLIC
russmckendrick / **jenkins-app** Updated a year ago	☆ 0	9	PUBLIC
russmckendrick / **jenkins** Updated a year ago	☆ 0	540	PUBLIC

Figure 3.3 – A list of my repositories in Docker

As you can see, there is a **Create Repository** button here. Since we are going to be looking at this process in more detail when it comes to the *Creating an automated build* section, we will move on to the next option.

Organizations

Organizations are those that you have either created or been added to. **Organizations** allow you to layer on control for, say, a project that multiple people are collaborating on. The organization gets its own settings, such as whether to store repositories as public or private by default or changing plans that will allow different numbers of private repositories and separate repositories altogether from the ones you or others have:



NAMESPACE	NAME	MY ROLE	TEAMS
masteringdockerfouthedition	Mastering Docker - Fourth Edition	Owner	1

Figure 3.4 – Viewing my list of organizations

You can also access or switch between accounts or organizations from the Dashboard just below the Docker logo, where you will typically see your username when you log in:

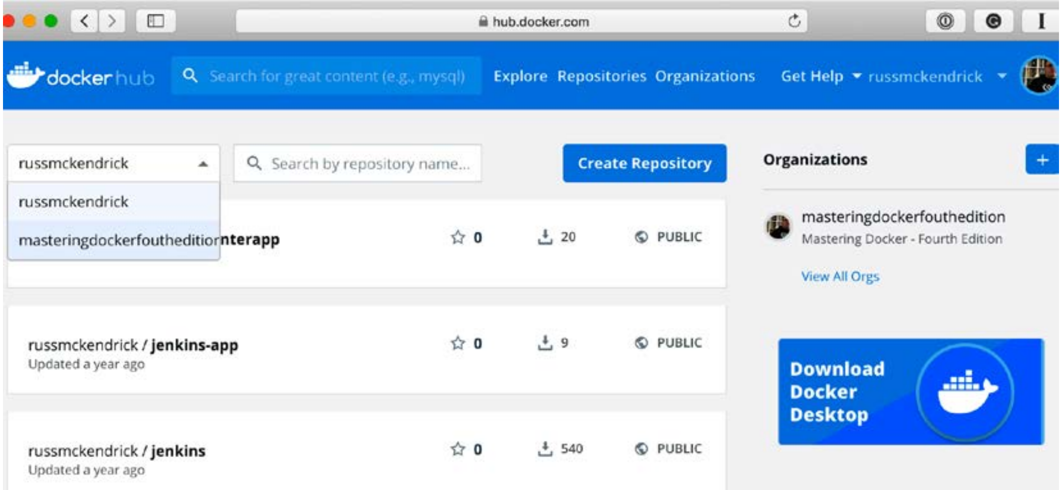


Figure 3.5 – Switching organizations

Organizations are useful when it comes to managing how you distribute your container images for different projects/applications.

Get Help

This dropdown menu is a jumping off point to various help and support sites managed by Docker. Let's take a quick look where each of these links will lead you to:

- **Documentation**: This takes you to the official documentation for Docker Hub.
- **Docker Hub**: This takes you straight to the Docker Hub category on the Docker community forums.
- **What's New**: This takes you to a list of Docker Blog posts tagged with Docker Hub.
- **Support**: This is an FAQ about Docker Hub with the option to contact support.

Profile and settings

The final option in the top menu is about managing your **Profile**, **Content**, and **Settings**:

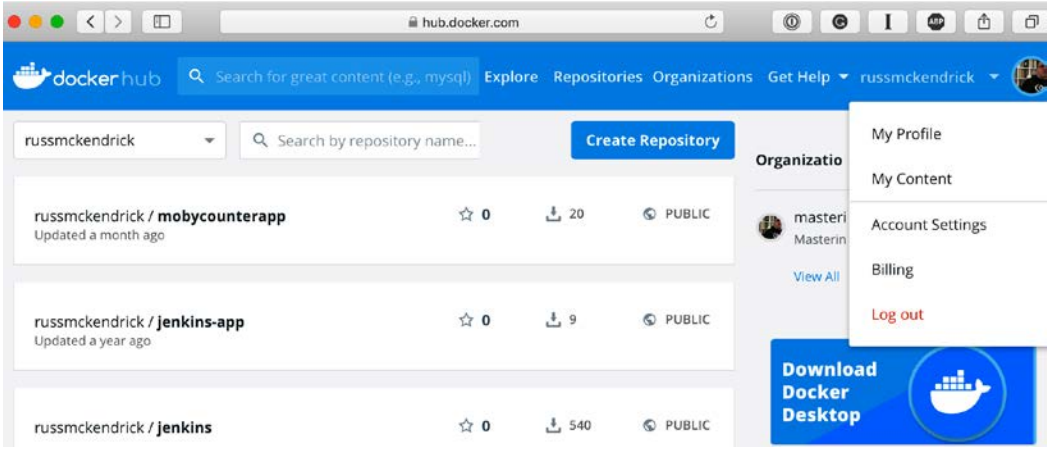


Figure 3.6 – Viewing your profile

The settings page allows you to set up your public profile, which includes the following options:

- **General**: You can add email addresses to your account, change your password, and configure what information is visible in your profile, such as your name, company location, and a link.
- **Linked Accounts**: Here, you can link your GitHub and Bitbucket accounts (more details on this will be discussed in the *Creating an automated build* section of this chapter).
- **Security**: This is where you can manage personal access tokens and the recently introduced Two-Factor authentication.
- **Default Privacy**: Do you want your newly created repositories to be public or private by default? This is where you can choose.
- **Notifications**: This is where you can sign up to notifications about your builds and account activity. Here, you can provide an email address or connect to a Slack installation.
- **Convert Account**: Here, you can convert your account into an organization. You probably don't want to do this; read the warnings on the page before going any further with this option.

- **Deactivate Account**: This does exactly what you would think it does. Again, take a look at the warnings on the page before doing anything as this action cannot be undone.
- **My Profile**: This menu item takes you to your public profile page; mine can be found at `https://hub.docker.com/u/russmckendrick/`.
- **My Content**: This link takes you to a list of containers that you may have subscribed to on Docker Hub.

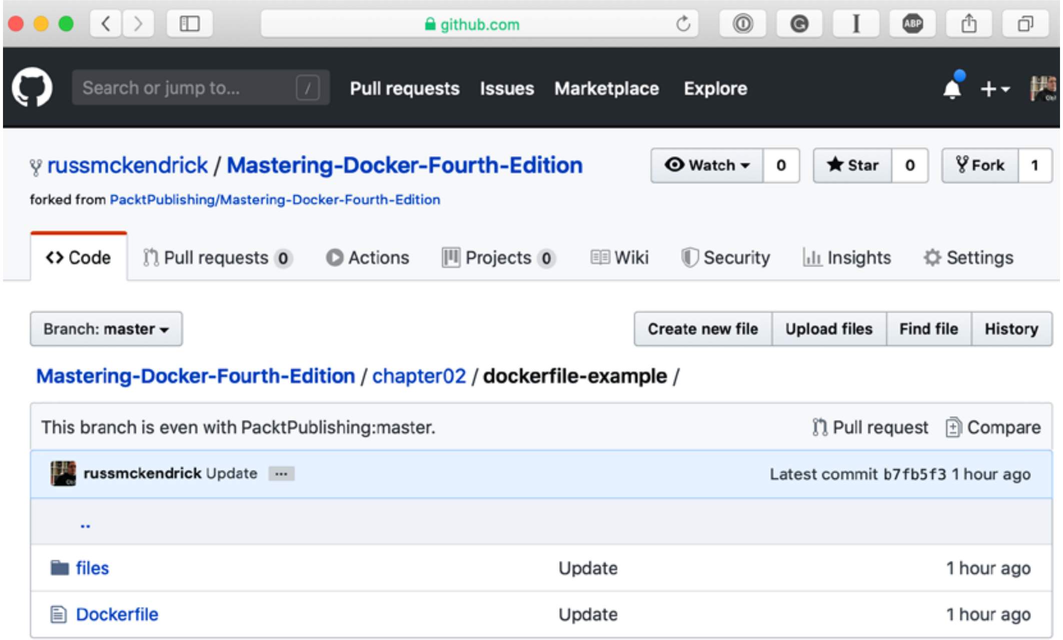
Creating an automated build

In this section, we will look at automated builds. Automated builds are those that you can link to your GitHub or Bitbucket account(s), and as you update the code in your code repository, you can have the image automatically built on Docker Hub. We will look at all the pieces required to do so. By the end of this section, you'll be able to automate your own builds.

Setting up your code

The first step to creating an automated build is to set up your GitHub or Bitbucket repository. These are the two options you have while selecting where to store your code. For example, I will be using GitHub, but the setup will be the same for you if you were using GitHub or Bitbucket.

In fact, I will be using the repository that accompanies this book. Since the repository is publicly available, you can fork it and follow along using your own GitHub account, as I have done in the following screenshot:



This branch is even with PacktPublishing:master.

russmckendrick Update … Latest commit b7fb5f3 1 hour ago

..		
files	Update	1 hour ago
Dockerfile	Update	1 hour ago

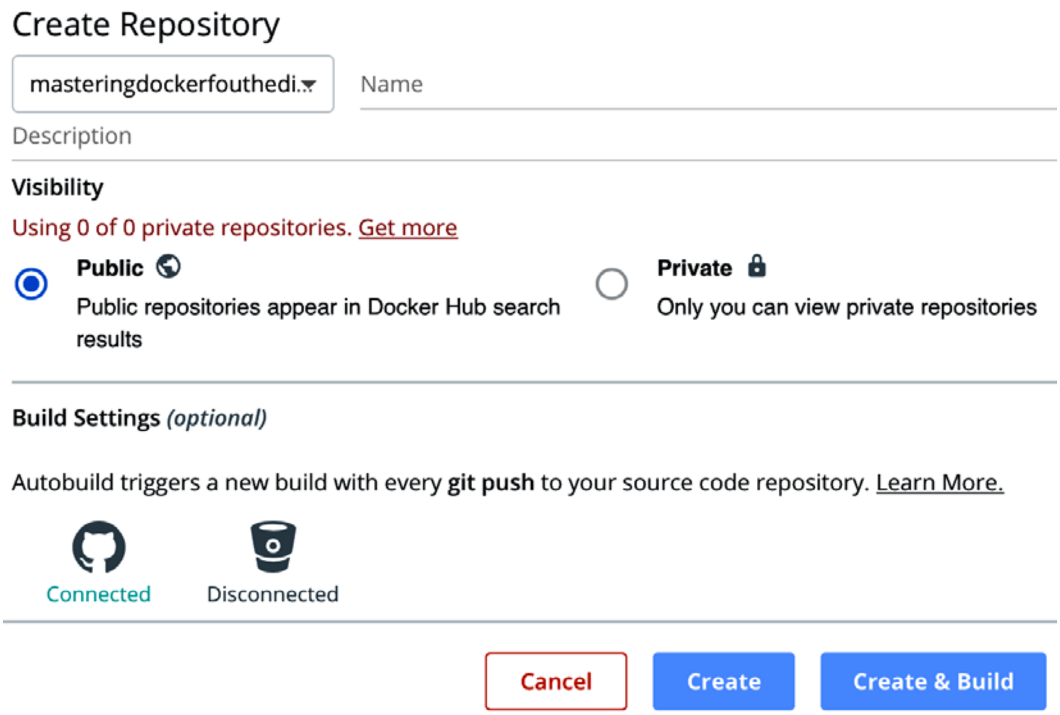
Figure 3.7 – Forking the accompanying GitHub repository

In *Chapter 2*, *Building Container Images*, we worked through a few different Dockerfiles. We will be using these for our automated builds. As you may recall, we installed NGINX and added a simple page with the message **Hello world! This is being served from Docker**. We also had a multi-stage build.

Now that we know which Dockerfile we will be using, let's set up a repository in Docker Hub.

Setting up Docker Hub

In **Docker Hub**, we are going to use the **Create Repository** button, which can be found under **Repositories**. After clicking it, we will be taken to a screen where we need to provide a little information about our build. We will also need to select a source:



Create Repository

masteringdockerfouthedi... Name

Description

Visibility

Using 0 of 0 private repositories. Get more

● Public — Public repositories appear in Docker Hub search results

○ Private — Only you can view private repositories

Build Settings *(optional)*

Autobuild triggers a new build with every **git push** to your source code repository. Learn More.

Figure 3.8 – Creating a repository in Docker Hub

As you can see from the preceding screenshot, I already have my GitHub account linked to my Docker Hub account. The process of linking the two tools was simple – all I had to do was allow Docker Hub permission to access my GitHub account by following the on-screen instructions.

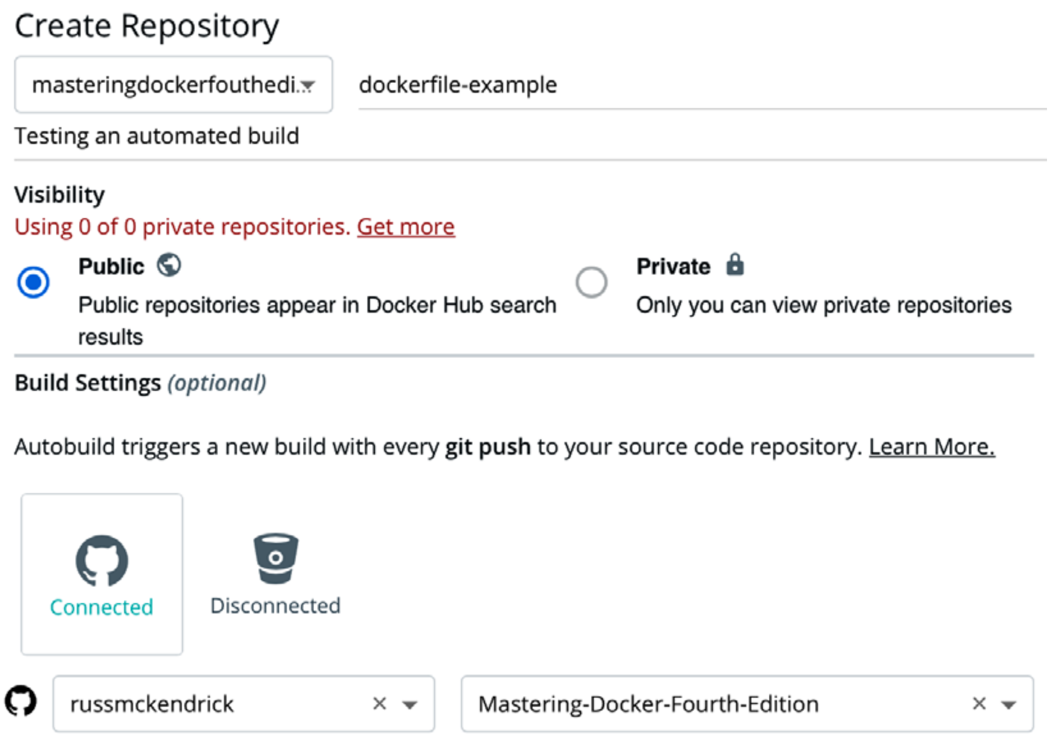
When connecting Docker Hub to GitHub, there are two options:

- **Public:** This limits Docker Hub's access to publicly available repositories and organizations. If you link your accounts using this option, Docker Hub won't be able to configure the web hooks needed for automated builds. You then need to search and select the repository from either of the locations you want to create the automated build from. This will essentially create a web hook that is triggered each time a commit is made on the selected GitHub repository. With this, a new build will be created on Docker Hub.

- **Private**: This is the recommended option out of the two as Docker Hub will have access to all your public and private repositories, as well as organizations. Docker Hub will also be able to configure the web hooks needed when setting up automated builds.

In the preceding screenshot, I selected `masteringdockerfourthedition` and visited the settings page for the automated build. From here, we can choose which Docker Hub profile the image is attached to, name the image, change it from a public to a privately available image, describe the build, and customize it by clicking on **Click here to customise**.

We can let Docker Hub know the location of our Dockerfile as follows:



Create Repository

masteringdockerfouthedi..▾ dockerfile-example

Testing an automated build

Visibility

Using 0 of 0 private repositories. Get more

● **Public**
Public repositories appear in Docker Hub search results

○ **Private**
Only you can view private repositories

Build Settings *(optional)*

Autobuild triggers a new build with every **git push** to your source code repository. Learn More.

russmckendrick × ▾ Mastering-Docker-Fourth-Edition × ▾

Figure 3.9 – This is what the completed form looks like

If you are following along, I entered the following information:

- **Repository Namespace and Name**: `dockerfile-example`
- **Short Description**: `Testing an automated build`
- **Visibility**: `Public`

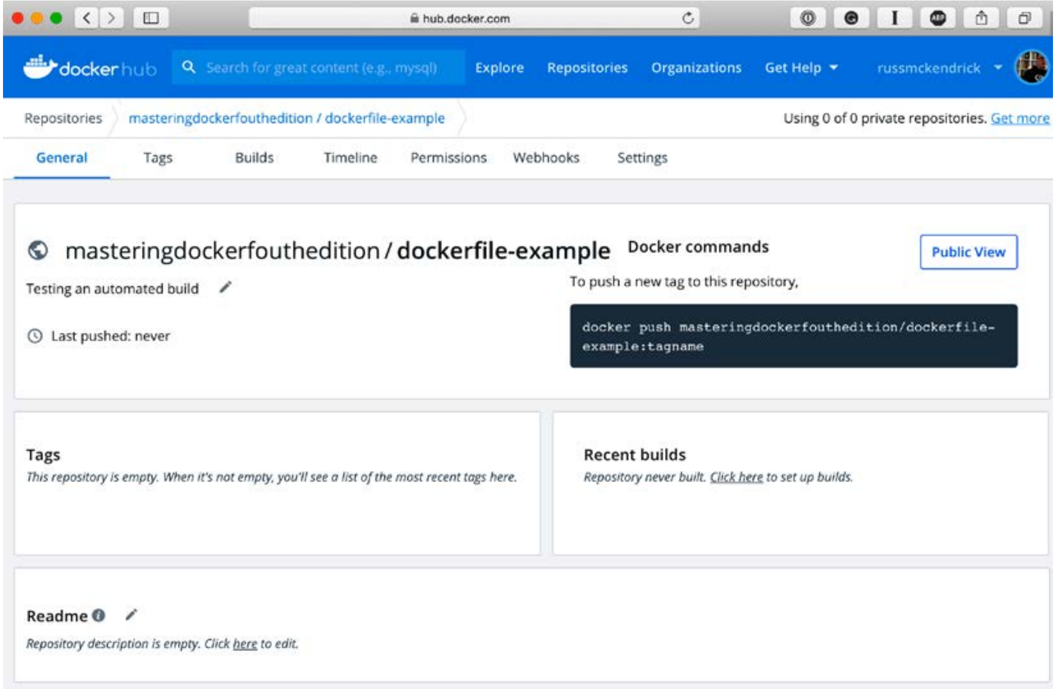
Then, under **Build Settings**, select **GitHub**:

- **Organization**: Select the GitHub organization where your code is hosted.
- **Repository**: If you have forked the code base that accompanies this book, enter `Master-Docker-Fourth-Edition`.

Click on the + icon next to Build Rules and then enter the following:

- **Source Type**: `Branch`
- **Source**: `master`
- **Docker Tag**: `latest`
- **Dockerfile Location**: `Dockerfile`
- **Build Caching**: Leave this selected

Upon clicking on **Create**, you will be taken to a screen similar to the following:



Using 0 of 0 private repositories. Get more

masteringdockerfouthedition / dockerfile-example

Testing an automated build

Last pushed: never

Docker commands

To push a new tag to this repository,

```
docker push masteringdockerfouthedition/dockerfile-
example:tagname
```

This repository is empty. When it's not empty, you'll see a list of the most recent tags here.

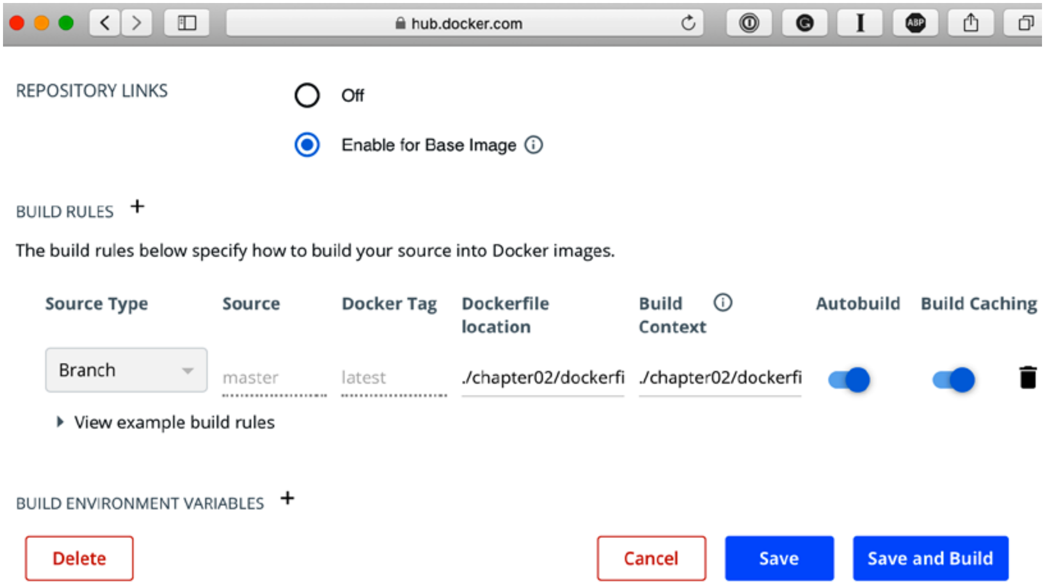
Repository never built. Click here to set up builds.

Repository description is empty. Click here to edit.

Figure 3.10 – The created repository

Now that we have our build defined, we can add some additional configurations by clicking on **Builds**. Since we are using the official Alpine Linux image in our Dockerfile, we can link that to our own build. When doing this, we also need to configure an additional path. To do this follow these steps:

1. Click on the **Configure Automated Builds** button. Then, click on the **Enable for Base Image** radio icon in the **Repository Links** section of the configuration and then the **Save** button.
2. This will kick off an unattended build each time a new version of the official Alpine Linux image is published.
3. Next, scroll down to **Build Rules**. For the **Build Context** setting, enter `./chapter02/dockerfile-example/`. This will make sure that Docker's build servers can find any files that we add to our Dockerfile:



REPOSITORY LINKS

○ Off

● Enable for Base Image ⓘ

BUILD RULES +

The build rules below specify how to build your source into Docker images.

Source Type	Source	Docker Tag	Dockerfile location	Build Context ⓘ	Autobuild	Build Caching
Branch	master	latest	./chapter02/dockerfi	./chapter02/dockerfi	●	●

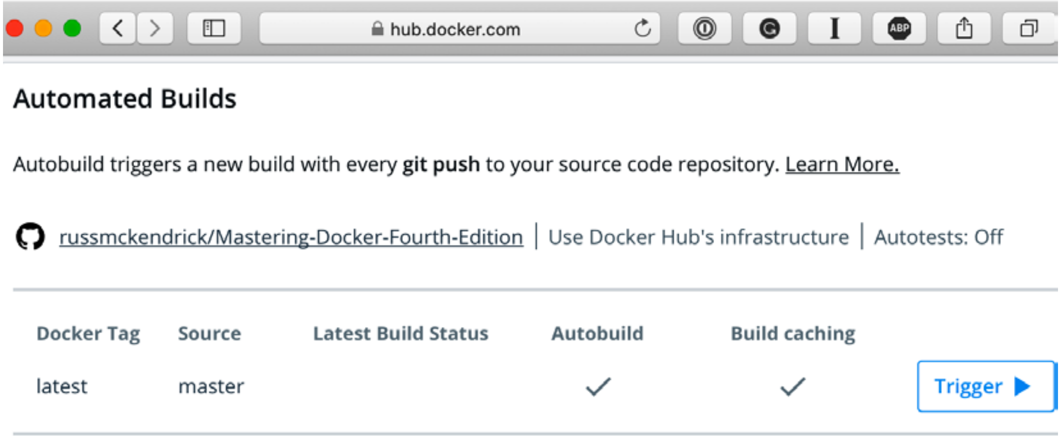
▸ View example build rules

BUILD ENVIRONMENT VARIABLES +

Figure 3.11 – Linking our repository to our Dockerfile

So, now, our image will be automatically rebuilt and published whenever we update the GitHub repository, or when a new official image is published.

As neither of these is likely to happen immediately, click on the **Trigger** button on the **Builds** page to manually kick off a build:



Automated Builds

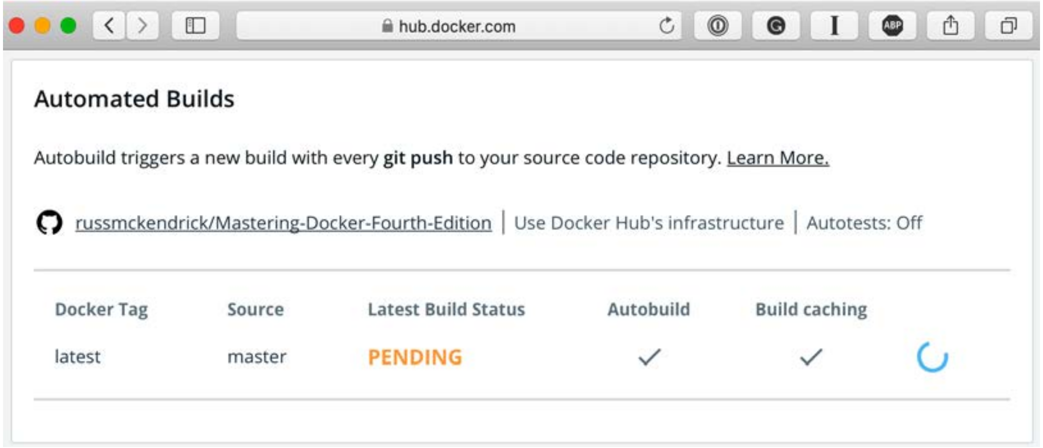
Autobuild triggers a new build with every **git push** to your source code repository. Learn More.

russmckendrick/Mastering-Docker-Fourth-Edition | Use Docker Hub's infrastructure | Autotests: Off

Docker Tag	Source	Latest Build Status	Autobuild	Build caching	
latest	master		✓	✓	Trigger ▶

Figure 3.12 – Triggering a build

You will notice that the **Trigger** button turns into a spinning icon and that **Latest Build Status** changes to **PENDING**, as shown in the following screenshot. This confirms that a build has been scheduled in the background:



Automated Builds

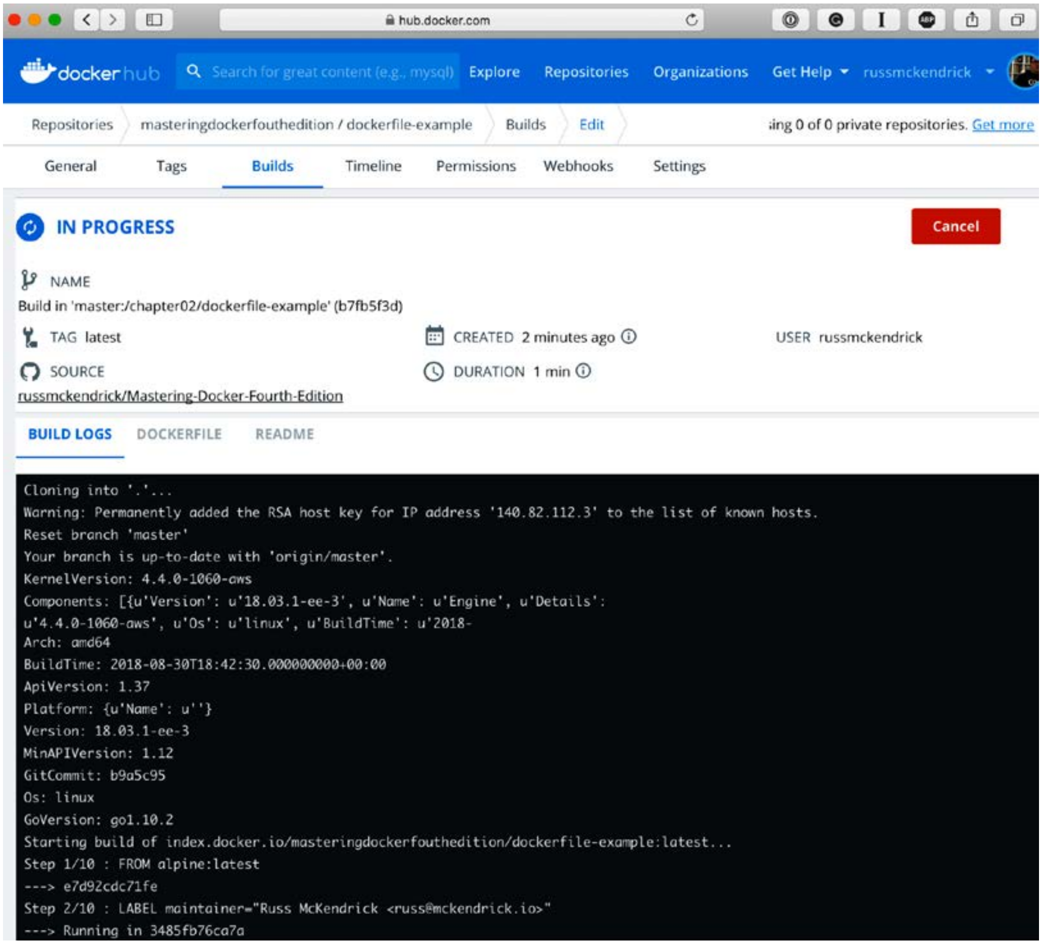
Autobuild triggers a new build with every **git push** to your source code repository. Learn More.

russmckendrick/Mastering-Docker-Fourth-Edition | Use Docker Hub's infrastructure | Autotests: Off

Docker Tag	Source	Latest Build Status	Autobuild	Build caching
latest	master	PENDING	✓	✓

Figure 3.13 – The build is progressing

Once you have triggered your build, scroll down to **Recent Builds**. This will list all of the builds for the image – successful, failed, and ones that are in progress. You should see a build underway; clicking on the build will bring up the logs for it:



NAME
Build in 'master:/chapter02/dockerfile-example' (b7fb5f3d)

TAG latest

CREATED 2 minutes ago

USER russmckendrick

SOURCE
russmckendrick/Mastering-Docker-Fourth-Edition

DURATION 1 min

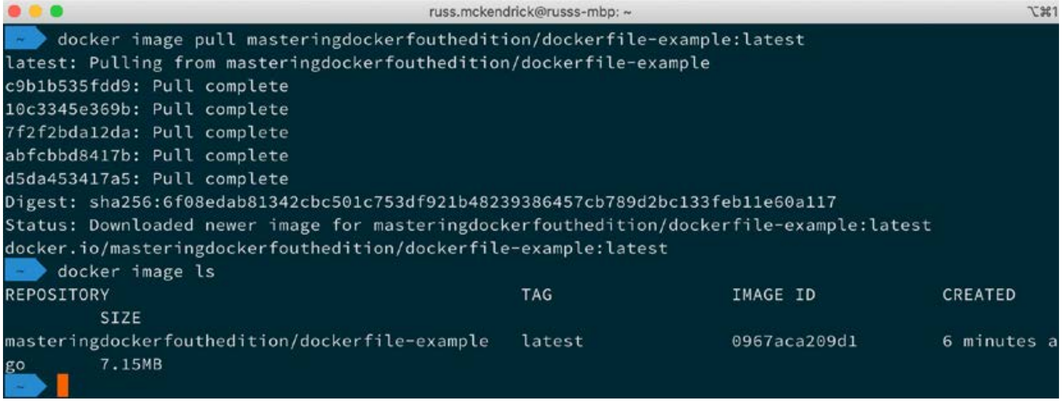
```
Cloning into '.'...
Warning: Permanently added the RSA host key for IP address '140.82.112.3' to the list of known hosts.
Reset branch 'master'
Your branch is up-to-date with 'origin/master'.
KernelVersion: 4.4.0-1060-aws
Components: [{u'Version': u'18.03.1-ee-3', u'Name': u'Engine', u'Details':
u'4.4.0-1060-aws', u'Os': u'linux', u'BuildTime': u'2018-
Arch: amd64
BuildTime: 2018-08-30T18:42:30.000000000+00:00
ApiVersion: 1.37
Platform: {u'Name': u''}
Version: 18.03.1-ee-3
MinAPIVersion: 1.12
GitCommit: b9a5c95
Os: linux
GoVersion: go1.10.2
Starting build of index.docker.io/masteringdockerfouthedition/dockerfile-example:latest...
Step 1/10 : FROM alpine:latest
---> e7d92cdc71fe
Step 2/10 : LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
---> Running in 3485fb76ca7a
```

Figure 3.14 – Viewing the progress of the build

Once built, you should then able to move to your local Docker installation by running the following commands, making sure to pull your own image if you have been following along:

```
$ docker image pull masteringdockerfouthedition/dockerfile-
example:latest
$ docker image ls
```

These commands are shown in the following screenshot:



```
russ.mckendrick@russs-mbp: ~
~ docker image pull masteringdockerfouthedition/dockerfile-example:latest
latest: Pulling from masteringdockerfouthedition/dockerfile-example
c9b1b535fdd9: Pull complete
10c3345e369b: Pull complete
7f2f2bda12da: Pull complete
abfcbbd8417b: Pull complete
d5da453417a5: Pull complete
Digest: sha256:6f08edab81342cbc501c753df921b48239386457cb789d2bc133feb11e60a117
Status: Downloaded newer image for masteringdockerfouthedition/dockerfile-example:latest
docker.io/masteringdockerfouthedition/dockerfile-example:latest
~ docker image ls
REPOSITORY                                      TAG                 IMAGE ID            CREATED
         SIZE
masteringdockerfouthedition/dockerfile-example  latest              0967aca209d1        6 minutes a
go       7.15MB
~
```

Figure 3.15 – Pulling our newly built image

You can also run the image created by Docker Hub using the following command, again making sure to use your own image if you have one:

```
$ docker container run -d -p8080:80 --name example
masteringdockerthirdedition/dockerfiles-example
```

I also add the multi-stage build in exactly the same way. Docker Hub had no problem with the build, as shown by the following logs, which start off with a little bit of information about Docker's build environment:

```
Cloning into '.'...
Warning: Permanently added the RSA host key for IP address
'140.82.114.3' to the list of known hosts.
Reset branch 'master'
Your branch is up-to-date with 'origin/master'.
KernelVersion: 4.4.0-1060-aws
Components: [{u'Version': u'18.03.1-ee-3', u'Name': u'Engine',
u'Details': {u'KernelVersion': u'4.4.0-1060-aws', u'Os':
u'linux', u'BuildTime': u'2018-08-30T18:42:30.000000000+00:00',
u'ApiVersion': u'1.37', u'MinAPIVersion': u'1.12',
u'GitCommit': u'b9a5c95', u'Arch': u'amd64', u'Experimental':
u'false', u'GoVersion': u'go1.10.2'}}]
Arch: amd64
BuildTime: 2018-08-30T18:42:30.000000000+00:00
ApiVersion: 1.37
Platform: {u'Name': u''}
Version: 18.03.1-ee-3
MinAPIVersion: 1.12
GitCommit: b9a5c95
Os: linux
GoVersion: go1.10.2
```

The build then starts by compiling our code, as follows:

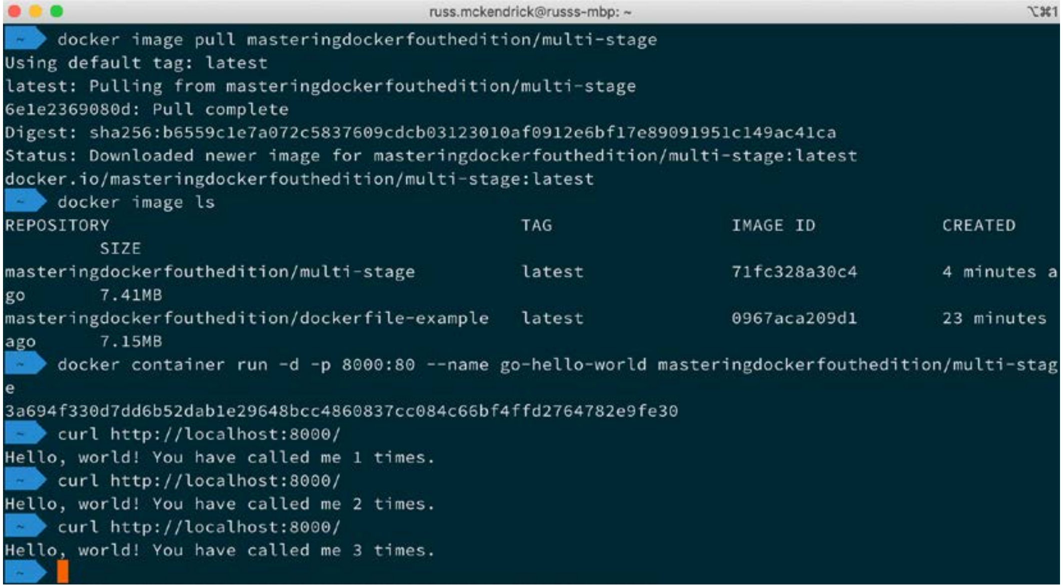
```
Starting build of index.docker.io/masteringdockerfouthedition/
multi-stage:latest...
Step 1/8 : FROM golang:latest as builder
---> 374d57ff6662
Step 2/8 : WORKDIR /go-http-hello-world/
Removing intermediate container 63fc21e72f2b
---> 25ed949838cf
Step 3/8 : RUN go get -d -v golang.org/x/net/html
---> Running in 57072354b296
get "golang.org/x/net/html": found meta tag get.
metaImport{Prefix:"golang.org/x/net", VCS:"git",
RepoRoot:"https://go.googlesource.com/net"} at //golang.org/x/
net/html?go-get=1
get "golang.org/x/net/html": verifying non-authoritative meta
tag
golang.org/x/net (download)
Removing intermediate container 57072354b296
---> 6731fc3ade79
Step 4/8 : ADD https://raw.githubusercontent.com/geetarista/
go-http-hello-world/master/hello_world/hello_world.go ./hello_
world.go
---> 2129f7e7cbab
Step 5/8 : RUN CGO_ENABLED=0 GOOS=linux go build -a
-installsuffix cgo -o app .
```

```
---> Running in 9d5646bf1b92
Removing intermediate container 9d5646bf1b92
---> 997b92d1a701
```

Now that our code has been compiled, it moves on to copying the application binary to what will be the final image using `scratch`:

```
Step 6/8 : FROM scratch
--->
Step 7/8 : COPY --from=builder /go-http-hello-world/app .
---> 70eb0af7f82c
Step 8/8 : CMD ["./app"]
---> Running in 41cc8b47f714
Removing intermediate container 41cc8b47f714
---> 71fc328a30c4
Successfully built 71fc328a30c4
Successfully tagged masteringdockerfouthedition/multi-
stage:latest
Pushing index.docker.io/masteringdockerfouthedition/multi-
stage:latest...
Done!
Build finished
You can pull and launch a container using the image with the
following commands:
$ docker image pull masteringdockerfouthedition/multi-stage
$ docker image ls
$ docker container run -d -p 8000:80 --name go-hello-world
masteringdockerfouthedition/multi-stage
$ curl http://localhost:8000/
```

As you can see from the following screenshot, the image acts in the exact same way as it did when we created it locally:



```
~ docker image pull masteringdockerfouthedition/multi-stage
Using default tag: latest
latest: Pulling from masteringdockerfouthedition/multi-stage
6e1e2369080d: Pull complete
Digest: sha256:b6559c1e7a072c5837609cdcb03123010af0912e6bf17e89091951c149ac41ca
Status: Downloaded newer image for masteringdockerfouthedition/multi-stage:latest
docker.io/masteringdockerfouthedition/multi-stage:latest
~ docker image ls
REPOSITORY                                   TAG                 IMAGE ID            CREATED
         SIZE
masteringdockerfouthedition/multi-stage      latest              71fc328a30c4        4 minutes a
go       7.41MB
masteringdockerfouthedition/dockerfile-example   latest              0967aca209d1        23 minutes
ago      7.15MB
~ docker container run -d -p 8000:80 --name go-hello-world masteringdockerfouthedition/multi-stag
e
3a694f330d7dd6b52dab1e29648bcc4860837cc084c66bf4ffd2764782e9fe30
~ curl http://localhost:8000/
Hello, world! You have called me 1 times.
~ curl http://localhost:8000/
Hello, world! You have called me 2 times.
~ curl http://localhost:8000/
Hello, world! You have called me 3 times.
~ 
```

Figure 3.16 – Pulling our multi-stage environment and launching the build

You can remove the containers if you launched them by using the following commands:

```
$ docker container stop example go-hello-world
$ docker container rm example go-hello-world
```

Now that we have looked at automated builds, we will discuss how else we can push images to Docker Hub.

Pushing your own image

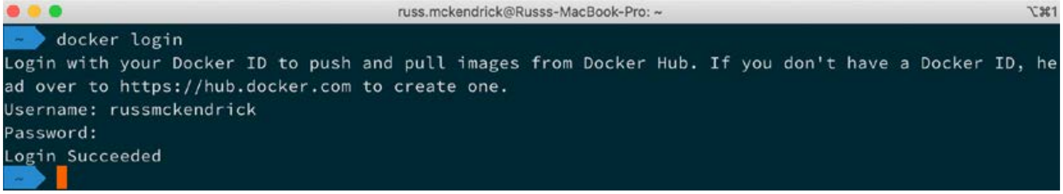
In *Chapter 2, Building Container Images*, we discussed creating an image without using a Dockerfile. While this is still not a good idea and should only be done when you really need to, you can push your own images to Docker Hub.

> **Tip**
>
> When pushing images to Docker Hub in this way, ensure that you do not include any code, files, or environment variables you would not want to be publicly accessible.

To do this, we first need to link our local Docker client to Docker Hub by running the following command:

```
$ docker login
```

You will then be prompted for your Docker ID and password. However, if you have enabled multi-factor authentication, then you will need to use a personal access token rather than your password. To create a personal access token, go to **Settings** in Docker Hub, click on **Security** from the left-hand menu, and then click the **New Access Token** button. As per the on-screen instructions, the access token will only be displayed once, so make sure you make a note of it. Treat personal access tokens as alternatives to your password and store them appropriately:



```
~ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, he
ad over to https://hub.docker.com to create one.
Username: russmckendrick
Password:
Login Succeeded
~
```

Figure 3.17 – Logging into Docker Hub using the Docker client

Also, if you are using Docker for Mac or Docker for Windows, you will now be logged in via the app and should be able to access Docker Hub from the menu:

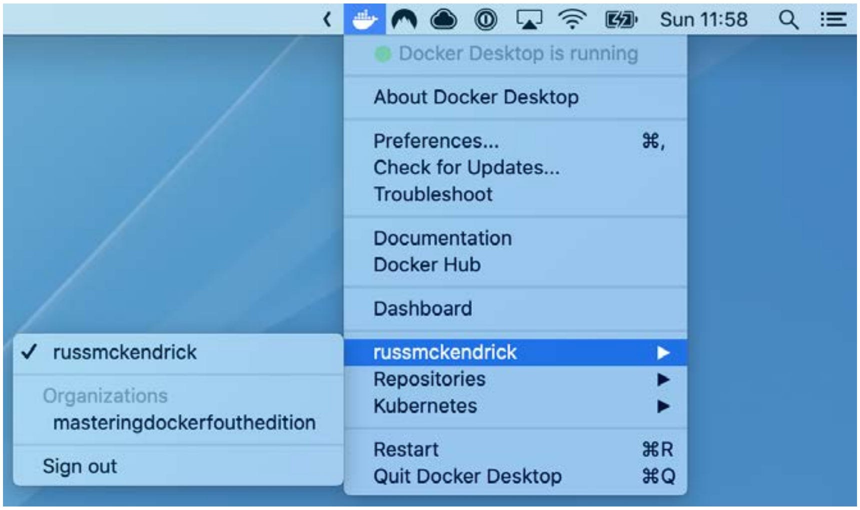


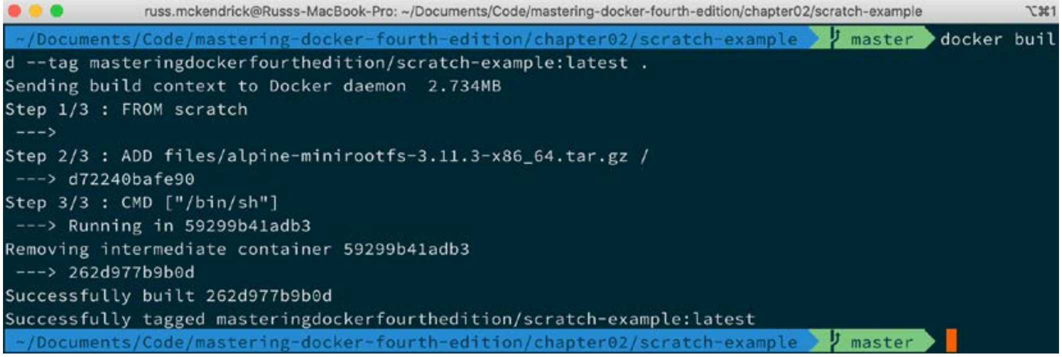
Figure 3.18 – Viewing your Docker Hub details in Docker Desktop

Now that our client is authorized to interact with Docker Hub, we need an image to build.

Let's look at pushing the scratch image we built in *Chapter 2*, *Building Container Images*. First, we need to build the image. To do this, I am using the following command:

```
$ docker build --tag masteringdockerfouthedition/scratch-example:latest .
```

If you are following along, then you should replace `masteringdockerfouthedition` with your own username or organization:



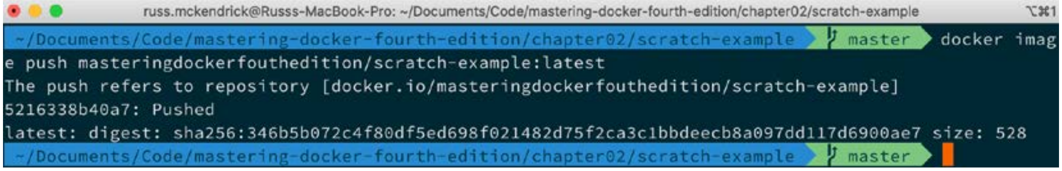
```
russ.mckendrick@Russs-MacBook-Pro: ~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example   ⌥⌘1
~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example  master  docker buil
d --tag masteringdockerfourthedition/scratch-example:latest .
Sending build context to Docker daemon  2.734MB
Step 1/3 : FROM scratch
 --->
Step 2/3 : ADD files/alpine-minirootfs-3.11.3-x86_64.tar.gz /
 ---> d72240bafe90
Step 3/3 : CMD ["/bin/sh"]
 ---> Running in 59299b41adb3
Removing intermediate container 59299b41adb3
 ---> 262d977b9b0d
Successfully built 262d977b9b0d
Successfully tagged masteringdockerfourthedition/scratch-example:latest
~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example  master
```

Figure 3.19 – Building an image locally

Once the image has been built, we can push it to Docker Hub by running the following command:

```
$ docker image push masteringdockerfouthedition/scratch-example:latest
```

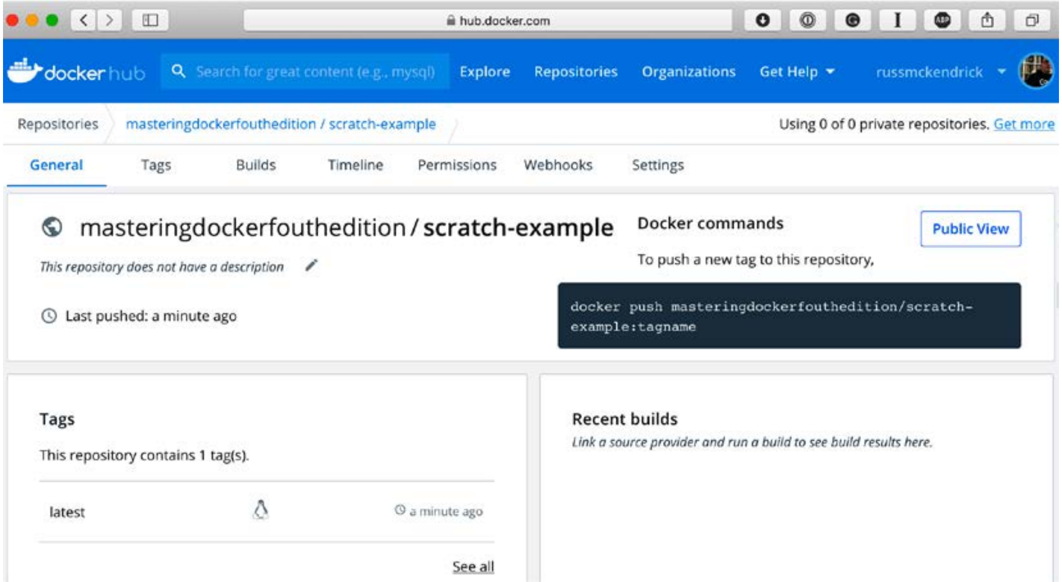
The following screenshot shows the output:



```
russ.mckendrick@Russs-MacBook-Pro: ~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example   ⌥⌘1
~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example  master  docker imag
e push masteringdockerfouthedition/scratch-example:latest
The push refers to repository [docker.io/masteringdockerfouthedition/scratch-example]
5216338b40a7: Pushed
latest: digest: sha256:346b5b072c4f80df5ed698f021482d75f2ca3c1bbdeecb8a097dd117d6900ae7 size: 528
~/Documents/Code/mastering-docker-fourth-edition/chapter02/scratch-example  master
```

Figure 3.20 – Pushing an image to Docker Hub

As you can see, because we defined `masteringdockerfouthedition/scratchexample:latest` when we built the image, Docker automatically uploaded the image to that location, which, in turn, added a new image to the `masteringdockerfouthedition` organization:



Using 0 of 0 private repositories. Get more

masteringdockerfouthedition / scratch-example

This repository does not have a description

Last pushed: a minute ago

Docker commands

To push a new tag to this repository,

```
docker push masteringdockerfouthedition/scratch-
example:tagname
```

This repository contains 1 tag(s).

Link a source provider and run a build to see build results here.

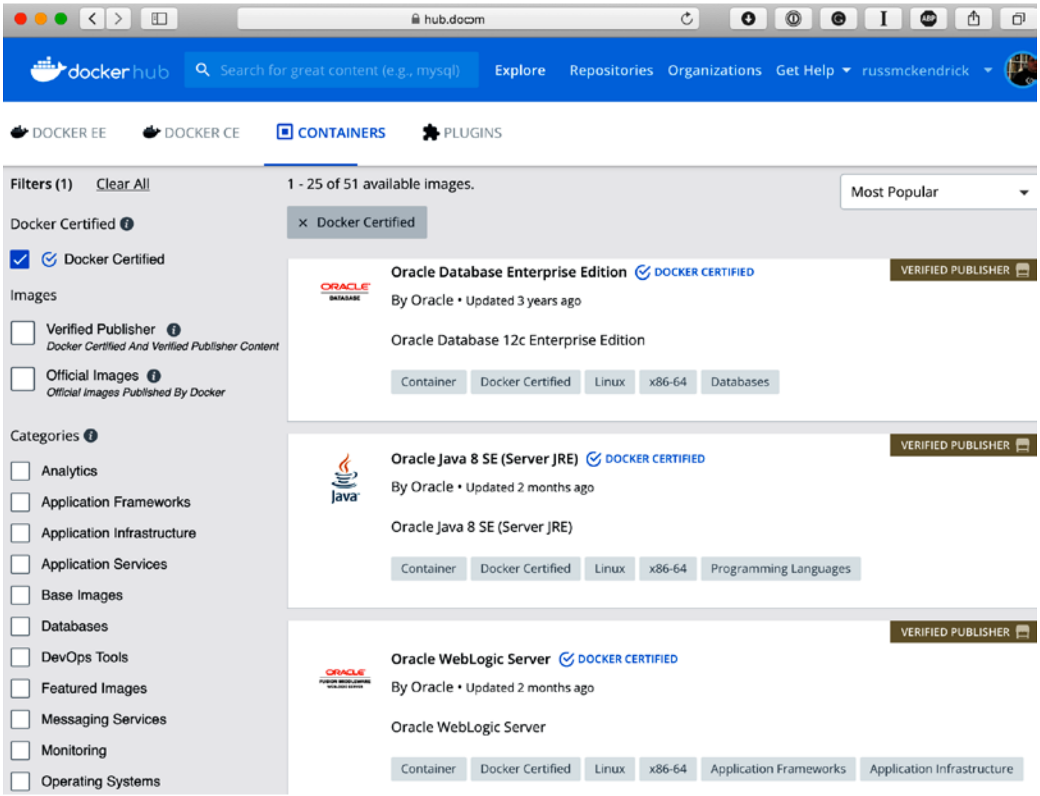
Figure 3.21 – Viewing our locally built image in Docker Hub

You will notice that there is not much you can do with the build in Docker Hub. This is because the image was not built by Docker Hub, and therefore, it does not really have any idea what has gone into building the image, which is exactly why this method of distributing images is discouraged.

Now that we have discussed how to distribute images, let's look at the complete opposite and discuss certified images and publishers.

Docker Certified Images and Verified Publishers

You may remember that in *Chapter 1*, *Docker Overview*, we downloaded Docker for macOS and Docker for Windows, as well as Docker Hub. As well as acting as a single location for downloading both Docker CE and Docker EE for various platforms, it is now also the preferred location for finding Docker Plugins, Docker Certified Images, and Images from Verified Publishers:



Filters (1) Clear All

1 - 25 of 51 available images.

Most Popular

× Docker Certified

Docker Certified

☑ Docker Certified

Images

□ Verified Publisher
Docker Certified And Verified Publisher Content

□ Official Images
Official Images Published By Docker

Categories

□ Analytics
□ Application Frameworks
□ Application Infrastructure
□ Application Services
□ Base Images
□ Databases
□ DevOps Tools
□ Featured Images
□ Messaging Services
□ Monitoring
□ Operating Systems

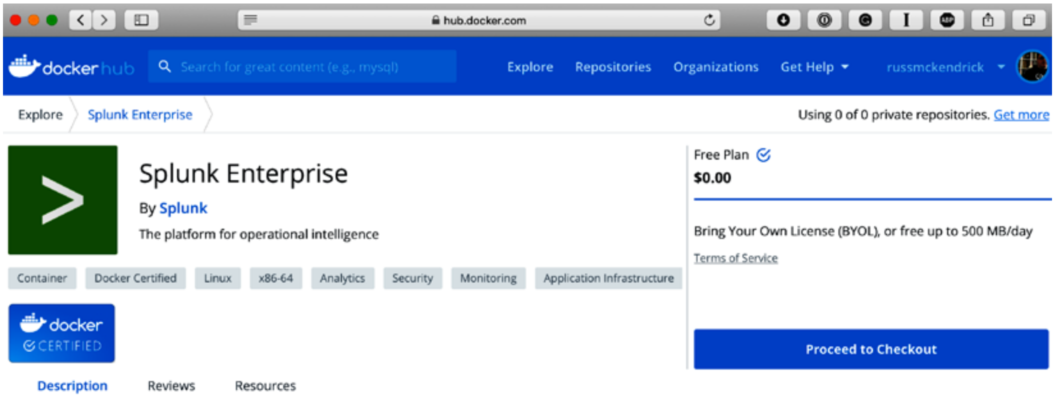
Oracle Database Enterprise Edition DOCKER CERTIFIED — VERIFIED PUBLISHER
By Oracle • Updated 3 years ago
Oracle Database 12c Enterprise Edition
Container | Docker Certified | Linux | x86-64 | Databases

Oracle Java 8 SE (Server JRE) DOCKER CERTIFIED — VERIFIED PUBLISHER
By Oracle • Updated 2 months ago
Oracle Java 8 SE (Server JRE)
Container | Docker Certified | Linux | x86-64 | Programming Languages

Oracle WebLogic Server DOCKER CERTIFIED — VERIFIED PUBLISHER
By Oracle • Updated 2 months ago
Oracle WebLogic Server
Container | Docker Certified | Linux | x86-64 | Application Frameworks | Application Infrastructure

Figure 3.22 – Exploring Docker Certified Images on Docker Hub

Taking a closer look at the **Splunk Enterprise** image in Docker Hub gives you information about who is responsible for the image. It also shows that it is a certified image, as shown in the following screenshot:



Explore > Splunk Enterprise

Using 0 of 0 private repositories. Get more

Splunk Enterprise
By Splunk
The platform for operational intelligence

Container | Docker Certified | Linux | x86-64 | Analytics | Security | Monitoring | Application Infrastructure

Free Plan
$0.00

Bring Your Own License (BYOL), or free up to 500 MB/day

Terms of Service

Figure 3.23 – Viewing the Splunk Enterprise Docker Hub image

As you may have noticed, the image has a price attached to it (the Free version is $0.00, but is limited), meaning that you can buy commercial software through Docker Hub since it has payments and licensing built in. If you are a software publisher, you can sign and distribute your own software through Docker Hub.

Deploying your own Docker Registry

In this section, we will look at Docker Registry. Docker Registry is an open source application that you can run anywhere you please and store your Docker image in. We will provide at a comparison between Docker Registry and Docker Hub, as well as how to choose between the two.

By the end of this section, you will have learned how to run your own Docker Registry and check whether it's a proper fit for you.

An overview of Docker Registry

Docker Registry, as stated earlier, is an open source application that you can utilize to store your Docker images on a platform of your choice. This allows you to keep them 100% private if you wish or share them as needed.

Docker Registry makes a lot of sense if you want to deploy your own registry without having to pay for all the private features of Docker Hub. Let's take a look at some comparisons between Docker Hub and Docker Registry to help you make an educated decision as to which option is best for you when it comes to choosing to store your own images.

Docker Registry has the following features:

- You can host and manage your own registry, from which you can serve all the repositories as private, public, or a mix between the two.
- You can scale the registry as needed, based on how many images you host or how many pull requests you are serving out.
- Everything is command-line-based.

With Docker Hub, you will be able to do the following:

- Get a GUI-based interface that you can use to manage your images
- Have a location already set up in the cloud that is ready to handle public and/or private images
- Have the peace of mind of not having to manage a server that is hosting all your images

Now that we know the differences between deploying our own registry and Docker Hub, let's look at the steps for deploying our own registry.

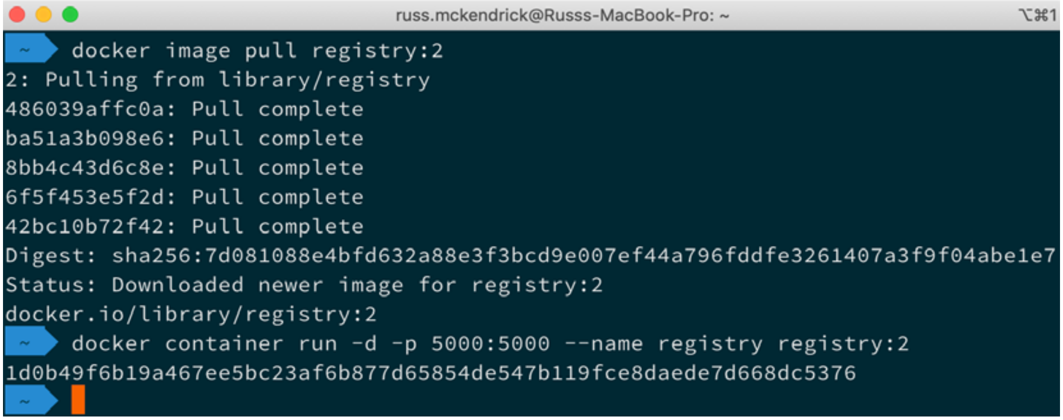
Deploying your own registry

As you may have already guessed, Docker Registry is distributed as an image from Docker Hub, which makes deploying it as easy as running the following commands:

```
$ docker image pull registry:2
```

```
$ docker container run -d -p 5000:5000 --name registry
registry:2
```

Running these commands should give you something like the following Terminal output:



```
~ docker image pull registry:2
2: Pulling from library/registry
486039affc0a: Pull complete
ba51a3b098e6: Pull complete
8bb4c43d6c8e: Pull complete
6f5f453e5f2d: Pull complete
42bc10b72f42: Pull complete
Digest: sha256:7d081088e4bfd632a88e3f3bcd9e007ef44a796fddfe3261407a3f9f04abe1e7
Status: Downloaded newer image for registry:2
docker.io/library/registry:2
~ docker container run -d -p 5000:5000 --name registry registry:2
1d0b49f6b19a467ee5bc23af6b877d65854de547b119fce8daede7d668dc5376
~
```

Figure 3.24 – Deploying your Docker Registry

These commands will give you the most basic installation of Docker Registry. Let's take a quick look at how we can push and pull an image to it. To start off with, we need an image, so let's grab the Alpine image (again):

```
$ docker image pull alpine
```

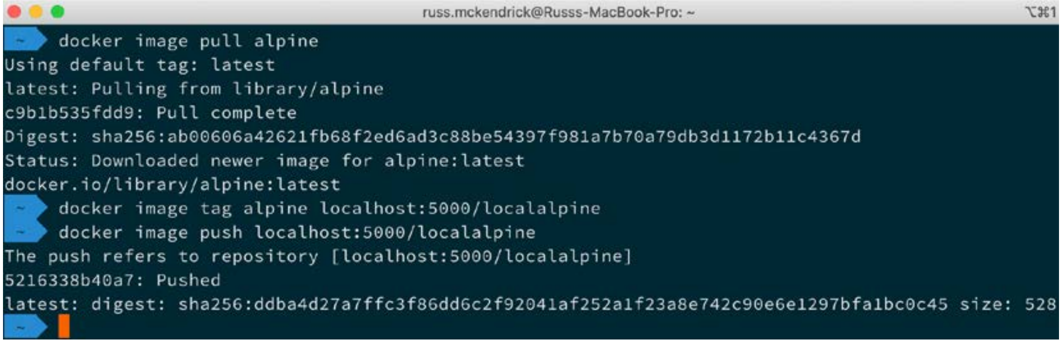
Now that we have a copy of the Alpine Linux image, we need to push it to our local Docker Registry, which is available at `localhost:5000`. To do this, we need to tag the Alpine Linux image with the URL of our local Docker Registry, along with a different image name:

```
$ docker image tag alpine localhost:5000/localalpine
```

Now that we have tagged our image, we can push it to our locally hosted Docker Registry by running the following command:

```
$ docker image push localhost:5000/localalpine
```

The following screenshot shows the output of the preceding commands:



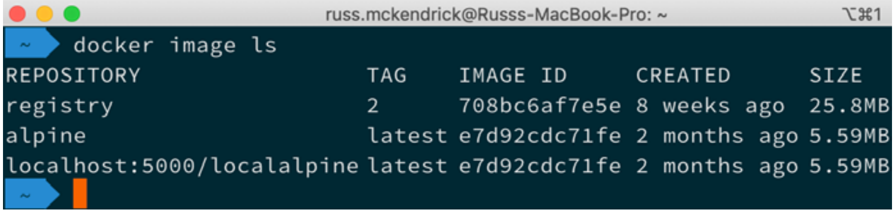
```
~  docker image pull alpine
Using default tag: latest
latest: Pulling from library/alpine
c9b1b535fdd9: Pull complete
Digest: sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b11c4367d
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
~  docker image tag alpine localhost:5000/localalpine
~  docker image push localhost:5000/localalpine
The push refers to repository [localhost:5000/localalpine]
5216338b40a7: Pushed
latest: digest: sha256:ddba4d27a7ffc3f86dd6c2f92041af252a1f23a8e742c90e6e1297bfa1bc0c45 size: 528
~
```

Figure 3.25 – Pushing an image to your own Docker Registry

Try running the following command:

```
$ docker image ls
```

The output should show you that you have two images with the same `IMAGE ID`:



```
~  docker image ls
REPOSITORY                  TAG    IMAGE ID     CREATED      SIZE
registry                    2      708bc6af7e5e 8 weeks ago  25.8MB
alpine                      latest e7d92cdc71fe 2 months ago 5.59MB
localhost:5000/localalpine latest e7d92cdc71fe 2 months ago 5.59MB
~
```

Figure 3.26 – Listing the images

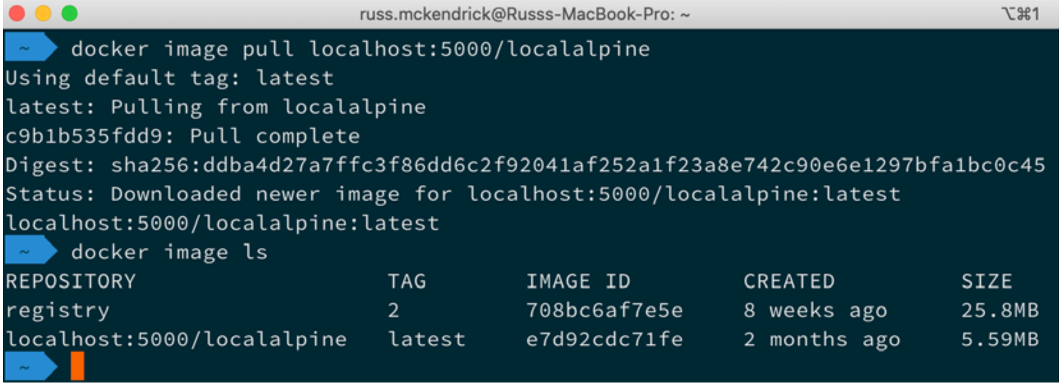
Before we pull the image back down from our local Docker Registry, we should remove the two local copies of the image. We need to use the `REPOSITORY` name to do this, rather than `IMAGE ID`, since we have two images from two locations with the same ID, and Docker will throw an error:

```
$ docker image rm alpine localhost:5000/localalpine
```

Now that the original and tagged images have been removed, we can pull the image from our local Docker Registry by running the following command:

```
$ docker image pull localhost:5000/localalpine
$ docker image ls
```

As you can see, we now have a copy of our image that has been pulled from the Docker Registry running at `localhost:5000`:



```
~  docker image pull localhost:5000/localalpine
Using default tag: latest
latest: Pulling from localalpine
c9b1b535fdd9: Pull complete
Digest: sha256:ddba4d27a7ffc3f86dd6c2f92041af252a1f23a8e742c90e6e1297bfa1bc0c45
Status: Downloaded newer image for localhost:5000/localalpine:latest
localhost:5000/localalpine:latest
~  docker image ls
REPOSITORY                   TAG       IMAGE ID        CREATED         SIZE
registry                     2         708bc6af7e5e    8 weeks ago     25.8MB
localhost:5000/localalpine   latest    e7d92cdc71fe    2 months ago    5.59MB
~
```

Figure 3.27 – Pulling from your own Docker Registry

You can stop and remove the Docker Registry by running the following commands:

```
$ docker container stop registry
$ docker container rm -v registry
```

Now, there are a lot of options and considerations when it comes to launching a Docker Registry. As you can imagine, the most important is in regards to storage.

Given that a registry's sole purpose is storing and distributing images, it is important that you use some level of persistent OS storage. Docker Registry currently supports the following storage options:

- **Filesystem**: This is exactly what its name suggests – all the images are stored on the filesystem at the path you define. The default is `/var/lib/registry`.
- **Azure**: This uses Microsoft Azure Blob Storage.
- **GCS**: This uses Google Cloud storage.
- **S3**: This uses **Amazon Simple Storage Service** (**Amazon S3**).
- **Swift**: This uses OpenStack Swift.

As you can see, other than the filesystem, all the storage engines that are supported are all highly available, distributed, object-level forms of storage.

Docker Trusted Registry

One of the components that ships with the commercial **Docker Enterprise Edition** (**Docker EE**) is **Docker Trusted Registry** (**DTR**), both of which are now being developed and supported by Mirantis. Think of it as a version of Docker Hub that you can host in your own infrastructure. DTR adds the following features on top of the ones provided by the free Docker Hub and Docker Registry:

- Integration into your authentication services, such as Active Directory or LDAP
- Deployment on your own infrastructure (or cloud) behind your firewall
- Image signing to ensure your images are trusted
- Built-in security scanning
- Access to prioritized support directly from Mirantis

Reviewing third-party registries

It is not only Docker that offers image registry services; companies such as Red Hat offer their own registry, where you can find the Red Hat Container Catalog, which hosts containerized versions of all of Red Hat's product offerings, along with containers to support its OpenShift offering. Services such as Artifactory by JFrog offer a private Docker registry as part of their build services.

There are also other Registry-as-a-Service offerings, such as Quay, which is also by Red Hat, as well as services from GitHub, Amazon Web Services, Microsoft Azure, and Google Cloud.

Let's take a quick look at some of these services.

GitHub Packages and Actions

The first service we are going to look at is GitHub Packages. Here, we will take a look at uploading a container to my fork of this book's GitHub repository. First of all, we are going to need a personal access token. To get this, log into your GitHub account and go to **Settings**, then **Developer settings**, and then **Personal access tokens**.

Generate an access token, call it `cli-package-access`, and give it the following permissions:

- `repo`
- `write:packages`
- `read:packages`
- `delete:packages`
- `workflow`

Make a note of the token when it is displayed as you will never be able to view it again. After doing this, I put my token in a file called `.githubpackage` in my users root folder. Putting it in there will mean that I don't need to enter the password each time I log in. I can do this by using the following command:

```
$ cat ~/.githubpackage | docker login docker.pkg.github.com -u
russmckendrick --password-stdin
```

Once logged in, we can build an image. For this example, I used `dockerfile-example`:

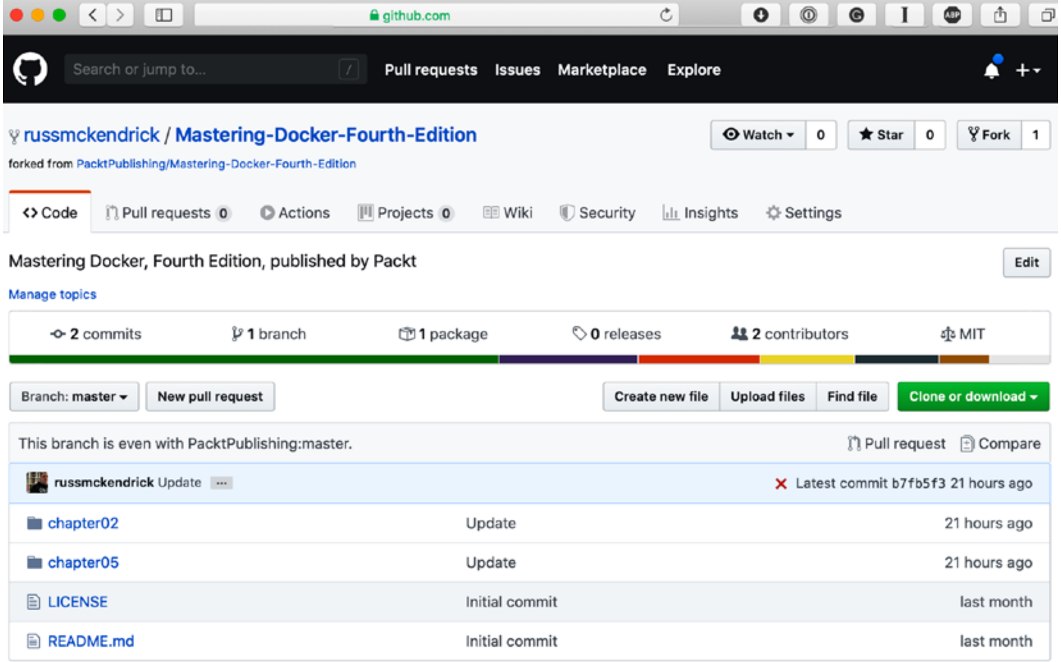
```
$ docker image build --tag docker.pkg.github.com/
russmckendrick/mastering-docker-fourth-edition/dockerfile-
example:latest .
```

Notice that I am using the repository name `mastering-docker-fourth-edition` and that it is all in lowercase. If you were to try and use any uppercase characters, Docker will complain.

Once built and tagged, you can push your image using the following command:

```
$ docker image push docker.pkg.github.com/russmckendrick/
mastering-docker-fourth-edition/dockerfile-example:latest
```

Once pushed, you should be able to see that there is now a package in your repository:



Mastering Docker, Fourth Edition, published by Packt

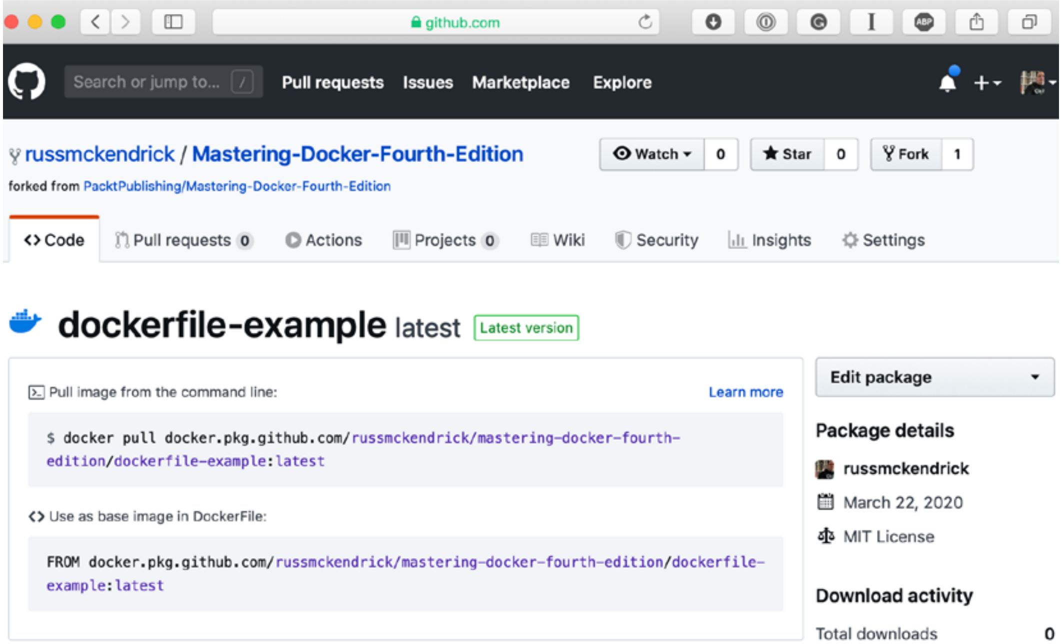
This branch is even with PacktPublishing:master.

russmckendrick Update ··· Latest commit b7fb5f3 21 hours ago

chapter02	Update	21 hours ago
chapter05	Update	21 hours ago
LICENSE	Initial commit	last month
README.md	Initial commit	last month

Figure 3.28 – Viewing the package that was pushed to GitHub

Drilling down to the package shows the following basic stats and download information:



dockerfile-example latest Latest version

Pull image from the command line:

```
$ docker pull docker.pkg.github.com/russmckendrick/mastering-docker-fourth-
edition/dockerfile-example:latest
```

Use as base image in DockerFile:

```
FROM docker.pkg.github.com/russmckendrick/mastering-docker-fourth-edition/dockerfile-
example:latest
```

Package details

russmckendrick

March 22, 2020

MIT License

Download activity

Total downloads	0

Figure 3.29 – Viewing more information about the package

You can also download the image by running the following command:

```
$ docker image pull docker.pkg.github.com/russmckendrick/
mastering-docker-fourth-edition/dockerfile-example:latest
```

Since our GitHub repository is public, our package will be too, meaning that anyone can download it.

So, that covers pushing an existing image. However, as we have mentioned a few times already throughout this chapter, this is not really recommended. Luckily, GitHub introduced GitHub Actions, which allows you to set up automated workflows that action **things** whenever an event, such as a push to the repository, occurs.

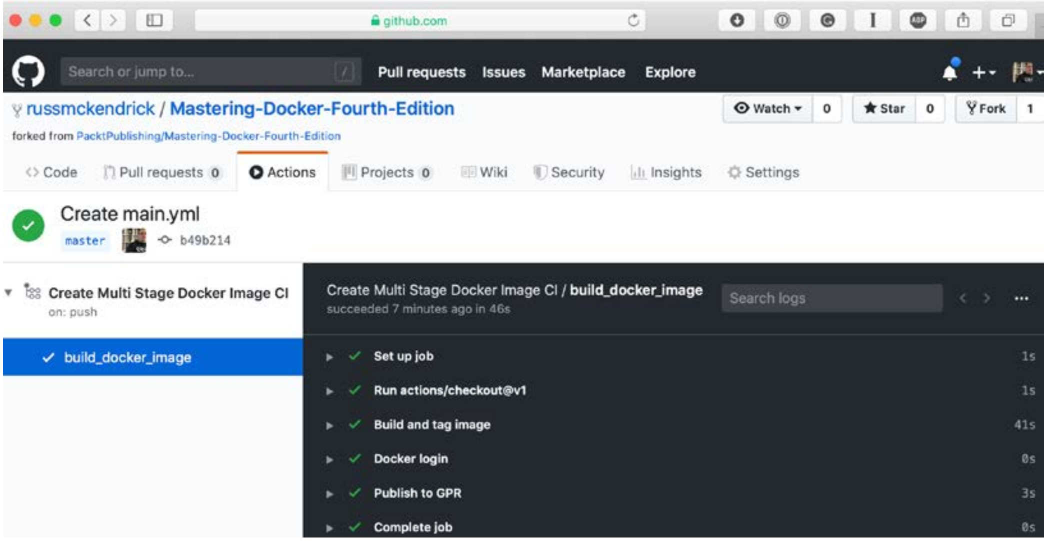
To create a GitHub Action, go to your repository and click on the **Actions** tab. Then, click on the **New workflow** button. This will default to a list of published Actions. Here, just click on the **Set up a workflow yourself** button; this will create a file called `.github/workflows/main.yml` in your repository.

Enter the following content in the space provided:

```
name: Create Multi Stage Docker Image CI

on: [push]

jobs:

  build_docker_image:

    runs-on: ubuntu-18.04

    steps:
    - uses: actions/checkout@v1

    - name: Build and tag image
      run: docker build -t "docker.pkg.github.com/$(echo
$GITHUB_REPOSITORY | tr '[:upper:]' '[:lower:]')/multi-
stage:$GITHUB_RUN_NUMBER" -f ./chapter02/multi-stage/Dockerfile
./chapter02/multi-stage/

    - name: Docker login
      run: docker login docker.pkg.github.com -u $GITHUB_ACTOR
-p $GITHUB_TOKEN
      env:
        GITHUB_TOKEN: ${{secrets.GITHUB_TOKEN}}

    - name: Publish to GPR
      run: docker push "docker.pkg.github.com/$(echo $GITHUB_
REPOSITORY | tr '[:upper:]' '[:lower:]')/multi-stage:$GITHUB_
RUN_NUMBER"
```

As you can see, this closely follows the steps we took to build and tag our image, authenticate against GitHub Packages, and push the image. There are some things that are relevant to GitHub Actions, such as the `$GITHUB_REPOSITORY` and `$GITHUB_RUN_NUMBER` variables, as well as `${{secrets.GITHUB_TOKEN}}`, all of which make sure that there is nothing from my Action that's hardcoded. This means you can run safely it in your own repository.

Once entered, click on the **Start commit** button, enter some details, and click on **Commit new file**. Once committed, the workflow will start automatically. You can view the output by going back to **Actions** and then selecting the newly created workflow and then job:



Create main.yml

master b49b214

Create Multi Stage Docker Image CI
on: push

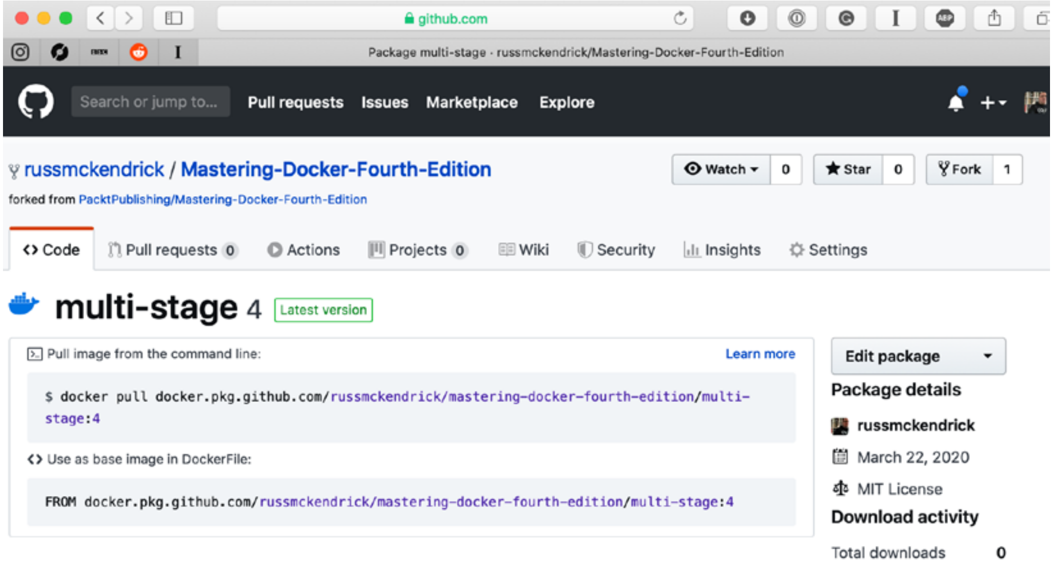
✓ build_docker_image

Create Multi Stage Docker Image CI / build_docker_image
succeeded 7 minutes ago in 46s

```
✓ Set up job                   1s
✓ Run actions/checkout@v1      1s
✓ Build and tag image         41s
✓ Docker login                 0s
✓ Publish to GPR               3s
✓ Complete job                 0s
```

Figure 3.30 – Viewing our GitHub Actions build results

Once complete, going to **Packages** in your repository should show you a package that looks as follows:


github.com
Package multi-stage · russmckendrick/Mastering-Docker-Fourth-Edition
Search or jump to...
Pull requests
Issues
Marketplace
Explore
russmckendrick / Mastering-Docker-Fourth-Edition
forked from PacktPublishing/Mastering-Docker-Fourth-Edition
Watch 0
Star 0
Fork 1
Code
Pull requests 0
Actions
Projects 0
Wiki
Security
Insights
Settings
Learn more
Edit package

multi-stage 4 Latest version

Pull image from the command line:

```
$ docker pull docker.pkg.github.com/russmckendrick/mastering-docker-fourth-edition/multi-stage:4
```

Use as base image in DockerFile:

```
FROM docker.pkg.github.com/russmckendrick/mastering-docker-fourth-edition/multi-stage:4
```

Package details

russmckendrick

March 22, 2020

MIT License

Download activity

Total downloads 0

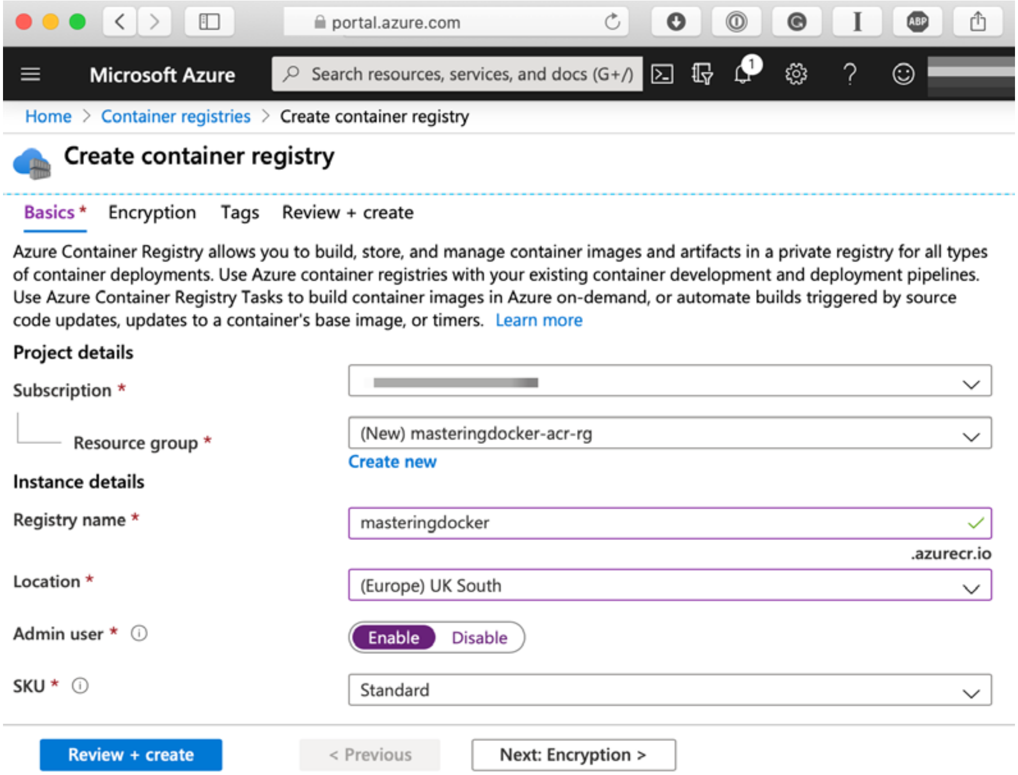
Figure 3.31 – Checking the package created by our GitHub Action

As you can see, while this does pretty much the same as an automated Docker Hub build, you have a lot more control over what happens and the build process itself.

Azure Container Registry

Next up on our third-party container registry walkthrough, we have Microsoft's Azure Container Registry. To create one, log into your Microsoft Azure account. We will talk about Microsoft Azure in more detail in *Chapter 10*, *Running Docker in Public Clouds*.

Once logged in, type `Container registries` into the search bar at the top of the screen and select the option from the results. Once the **Container registries** page loads, click on the + **Add** button. You will be presented with a page that looks as follows:


portal.azure.com
Microsoft Azure
Search resources, services, and docs (G+/)
Home > Container registries > Create container registry
Create container registry
Basics * | Encryption | Tags | Review + create
Review + create
< Previous
Next: Encryption >

Azure Container Registry allows you to build, store, and manage container images and artifacts in a private registry for all types of container deployments. Use Azure container registries with your existing container development and deployment pipelines. Use Azure Container Registry Tasks to build container images in Azure on-demand, or automate builds triggered by source code updates, updates to a container's base image, or timers. Learn more

Project details

Subscription *: [redacted]

Resource group *: (New) masteringdocker-acr-rg (Create new)

Instance details

Registry name *: masteringdocker .azurecr.io

Location *: (Europe) UK South

Admin user *: Enable | Disable

SKU *: Standard

Figure 3.32 – Creating our Azure Container Registry

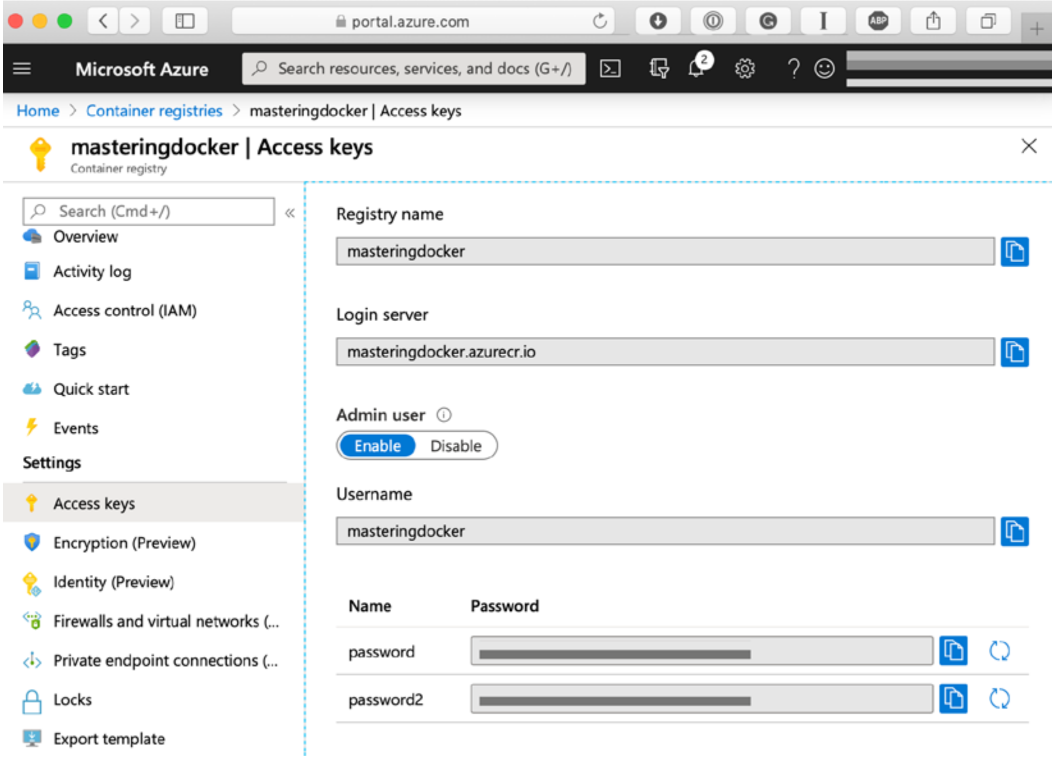
If you are following along, enter the following information:

- **Subscription**: Select the subscription you would like to use.
- **Resource Group**: Select or create the resource group you would like to host your registry. I created one called `masteringdocker-acr-rg`.

- **Registry Name**: Choose a name for your registry. This needs to be unique. I used `masteringdocker`.
- **Location**: Where do you want your container registry to be hosted? I choose `UK South`.
- **Admin User**: Select `Enable`.
- **SKU**: Choose `Basic`. This should be enough for testing.

We are going to ignore the encryption options for now as they are only available when using the premium SKU as well as the tags, so click on **Review + Create**. Once your deployment has been validated, click on the **Create** button. After a few minutes, your deployment will be complete.

As with GitHub Packages, we are going to build and push a container. To do this, we need some credentials. To find these, click on **Access Keys** and make a note of the details for **Login server**, **Username**, and one of the two **passwords**:



Registry name: masteringdocker

Login server: masteringdocker.azurecr.io

Admin user: [✓ Enable] Disable

Username: masteringdocker

Name	Password
password	[redacted]
password2	[redacted]

Figure 3.33 – Getting the access key for our Azure Container Registry

Like with GitHub Packages, put the password in a text file. I used `~/.azureacrpassword`. Then, log in with the Docker command-line client by running the following:

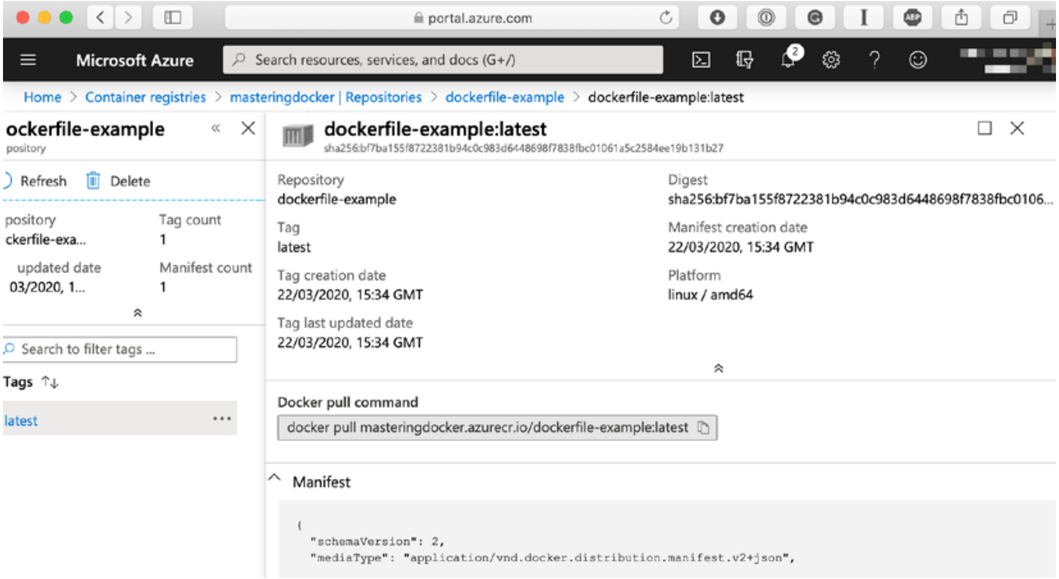
```
$ cat ~/.azureacrpassword | docker login masteringdocker.
azurecr.io -u masteringdocker --password-stdin
```

Now that we are authenticated, change to the `dockerfile-example` folder, which can be found in the `chapter02` folder in this book's GitHub repository, and build, tag, and push our image:

```
$ docker image build --tag masteringdocker.azurecr.io/
dockerfile-example:latest .
```

```
$ docker image push masteringdocker.azurecr.io/dockerfile-
example:latest
```

Once pushed, you should be able to see it listed on the Azure Container Registry page by clicking on **Registries** in the **Services** section of the main menu. Select the image and version. After doing this, you will see something like the following:


portal.azure.com
Microsoft Azure
Search resources, services, and docs (G+/)
Home > Container registries > masteringdocker | Repositories > dockerfile-example > dockerfile-example:latest
ockerfile-example
pository
Refresh
Delete
Search to filter tags ...
Tags
latest
Manifest

dockerfile-example:latest

sha256:bf7ba155f8722381b94c0c983d6448698f7838fbc01061a5c2584ee19b131b27

pository: ckerfile-exa...

Tag count: 1

updated date: 03/2020, 1...

Manifest count: 1

Repository: dockerfile-example

Digest: sha256:bf7ba155f8722381b94c0c983d6448698f7838fbc0106...

Tag: latest

Manifest creation date: 22/03/2020, 15:34 GMT

Tag creation date: 22/03/2020, 15:34 GMT

Platform: linux / amd64

Tag last updated date: 22/03/2020, 15:34 GMT

Docker pull command

```
docker pull masteringdocker.azurecr.io/dockerfile-example:latest
```

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
```

Figure 3.34 – Viewing our container in the Azure Container Registry

When it comes to pulling images, you will need to make sure that you are authenticated against your Container Registry as it is a private service. Trying to pull and not being logged in will result in an error.

You can also automate these builds based on committing your Dockerfile to a GitHub repository. This is, however, a little more involved as it can currently only be configured using Azure's command-line tools. See the *Further reading* section for more information about how to configure Azure Container Registry Tasks.

Looking at MicroBadger

Microbadger is a great tool when you are looking at shipping your containers or moving your images around. It will take into account everything that is going on in every single layer of a particular Docker image and give you an output regarding how much weight it has in terms of its actual size or the amount of disk space it will take up.

The following page is what you will be presented with when navigating to the MicroBadger website, `https://microbadger.com/`:

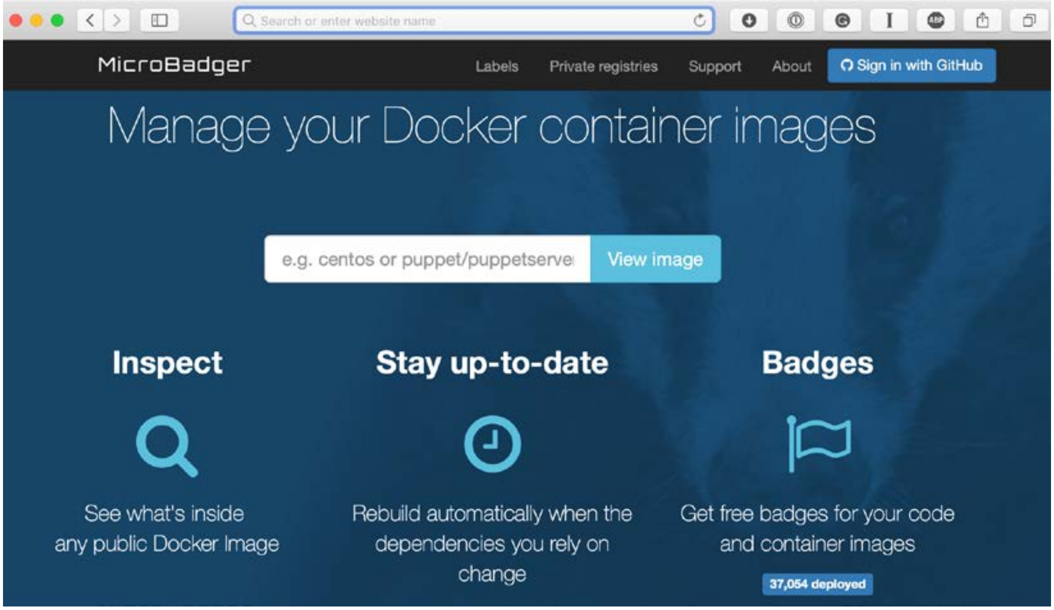
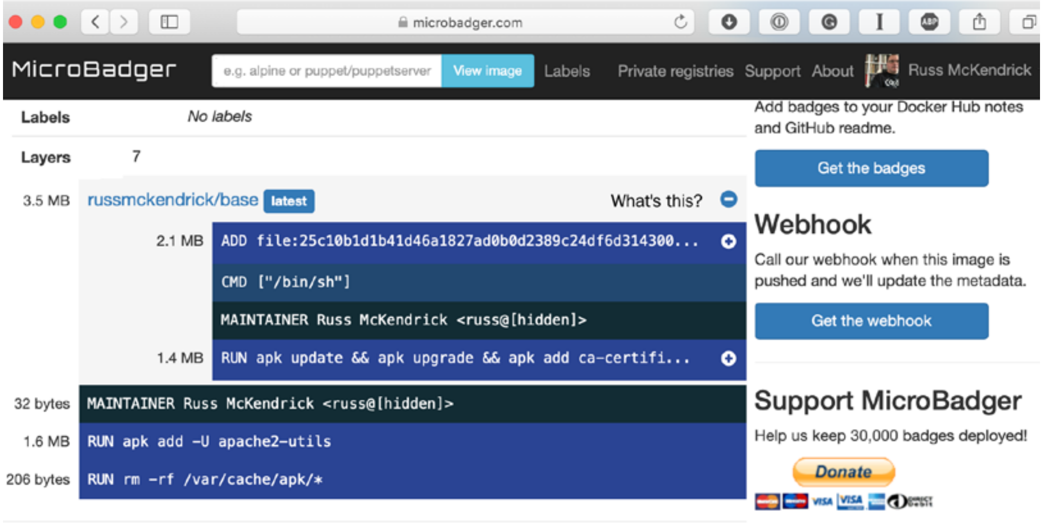

Search or enter website name
MicroBadger
Labels
Private registries
Support
About
Sign in with GitHub
Manage your Docker container images
e.g. centos or puppet/puppetserve
View image
Inspect
See what's inside any public Docker Image
Stay up-to-date
Rebuild automatically when the dependencies you rely on change
Badges
Get free badges for your code and container images
37,054 deployed

Figure 3.35 – The Microbadger home page

You can search for images that are on Docker Hub to have MicroBadger provide information about that image. Alternatively, you can load up a sample image set if you are looking to provide some sample sets, or to view some more complex setups.

In this example, we are going to search for one of my images, `russmckendrick/ab`, and select the latest tag. By default, it will always load the latest tag, but you also have the option of changing the tag you are viewing by selecting your desired tag from the **Versions** drop-down menu. This could be useful if you have, for example, a staging tag and are thinking of pushing this new image to your latest tag, but want to see what impact it will have on the size of the image.

As shown in the following screenshot, MicroBadger presents information about how many layers your image contains:



Labels No labels

Layers 7

3.5 MB russmckendrick/base latest What's this?

```
2.1 MB    ADD file:25c10b1d1b41d46a1827ad0b0d2389c24df6d314300...
          CMD ["/bin/sh"]
          MAINTAINER Russ McKendrick <russ@[hidden]>
1.4 MB    RUN apk update && apk upgrade && apk add ca-certifi...
32 bytes  MAINTAINER Russ McKendrick <russ@[hidden]>
1.6 MB    RUN apk add -U apache2-utils
206 bytes RUN rm -rf /var/cache/apk/*
```

Add badges to your Docker Hub notes and GitHub readme.

Webhook

Call our webhook when this image is pushed and we'll update the metadata.

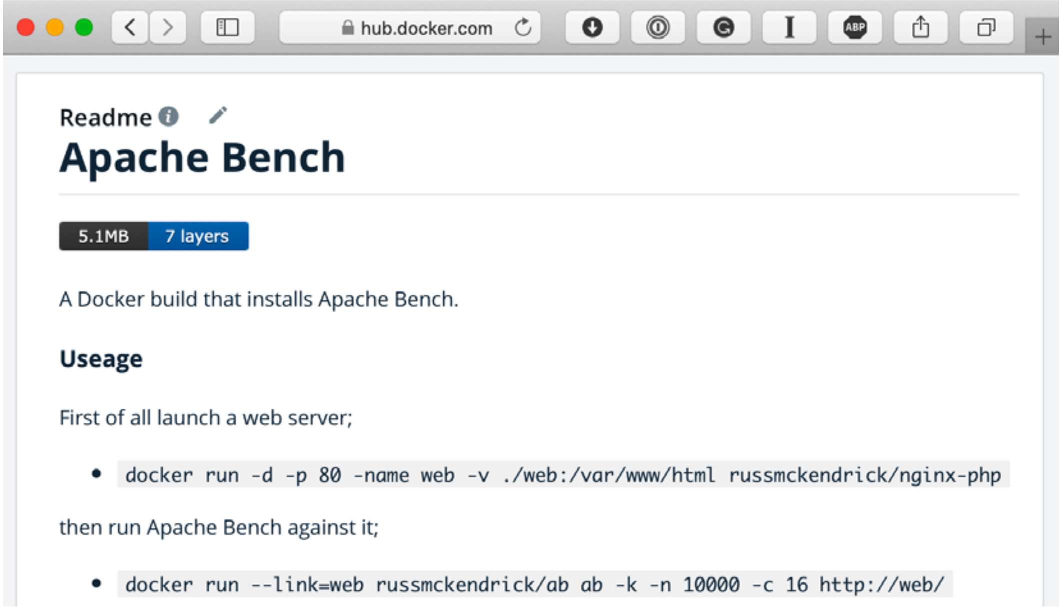
Support MicroBadger

Help us keep 30,000 badges deployed!

Figure 3.36 – Viewing the details of our container in MicroBadger

By showing the size of each layer and the Dockerfile command that was executed during the image build, you can see at which stage of the image build the bloat was added, which is extremely useful when it comes to reducing the size of your images.

Another great feature of MicroBadger is that it gives you the option to embed basic statistics about your images in your GitHub repository or Docker Hub. For example, the following screenshot shows the Docker Hub page for `russmckendrick/ab`:



Readme

Apache Bench

5.1MB 7 layers

A Docker build that installs Apache Bench.

Useage

First of all launch a web server;

- `docker run -d -p 80 -name web -v ./web:/var/www/html russmckendrick/nginx-php`

then run Apache Bench against it;

- `docker run --link=web russmckendrick/ab ab -k -n 10000 -c 16 http://web/`

Figure 3.37 – Adding MicroBadger stats to an image's README file

As you can see, MicroBadger is displaying the overall size of the image, which in this example is `5.1MB`, as well as the total number of layers the image is made up of, which is `7`. The MicroBadger service is still in its beta stage and new functions are being added all the time. I recommend that you keep an eye on it.

Summary

In this chapter, we looked at several ways in which we can both manually and automatically build container images using Docker Hub. We discussed the various registries we can use besides Docker Hub, such as GitHub Packages and Microsoft's Azure Container Registry.

We also looked at deploying our own local Docker Registry and touched upon the considerations we need to make around storage when deploying one. Finally, we looked at MicroBadger, a service that allows us to display information about our remotely hosted container images.

All of this means you now have a way of distributing your own container images, both securely and in a way that allows you to easily keep your container images up to date.

This is important as it means that, if you wish, it is possible to trigger an update of all your images with a single build, rather than having to manually build and push each individual image.

In the next chapter, we are going to look at how to manage our containers from the command line.

Questions

1. True or false: Docker Hub is the only source from which you can download official Docker images.
2. Describe why you would want to link an automated build to an official Docker Hub image.
3. Are multi-stage builds supported on Docker Hub?
4. True or false: Logging into Docker on the command line also logs you into the desktop application.
5. How would you delete two images that share the same `IMAGE ID`?
6. Which port does Docker Registry run on by default?

Further reading

More information on Docker Store, Trusted Registry, and Registry can be found at the following links:

- Docker Hub Publisher Signup: `https://store.docker.com/publisher/signup/`
- Docker Registry Documentation: `https://docs.docker.com/registry/`
- **Docker Trusted Registry** (**DTR**): `https://www.mirantis.com/software/docker/image-registry/`

You can find more details about the different types of cloud-based storage you can use for Docker Registry at the following links:

- Azure Blob Storage: `https://azure.microsoft.com/en-gb/services/storage/blobs/`
- Google Cloud storage: `https://cloud.google.com/storage/`

- **Amazon Simple Storage Service** (**Amazon S3**): `https://aws.amazon.com/s3/`
- Swift: `https://wiki.openstack.org/wiki/Swift`

Some of the third-party registry services can be found here:

- GitHub Actions: `https://github.com/features/actions`
- Azure Container Registry: `https://azure.microsoft.com/en-gb/services/container-registry/`
- Azure Container Registry Tasks: `https://docs.microsoft.com/en-gb/azure/container-registry/container-registry-tutorial-quick-task`
- Amazon Elastic Container Registry: `https://aws.amazon.com/ecr/`
- Google Cloud Container Registry: `https://cloud.google.com/container-registry`
- Red Hat Container Catalog: `https://catalog.redhat.com/software/containers/explore`
- OpenShift: `https://www.openshift.com/`
- Quay by Red Hat: `https://quay.io/`
- Artifactory by JFrog: `https://www.jfrog.com/artifactory/`

Finally, you can find links to Docker Hub and Microbadger for my Apache Bench image here:

- Apache Bench Image (Docker Hub): `https://hub.docker.com/r/russmckendrick/ab/`
- Apache Bench Image (Microbadger): `https://microbadger.com/images/russmckendrick/ab`

4
Managing Containers

So far, we have been concentrating on how to build, store, and distribute our Docker images. Now we are going to look at how we can launch containers, and also how we can use the Docker command-line client to manage and interact with them.

We will be revisiting the commands we used in *Chapter 1*, *Docker Overview*, by going into a lot more detail, before delving deeper into the commands that are available. Once we are familiar with the container commands, we will look at Docker networks and Docker volumes.

We will cover the following topics in this chapter:

- Understanding Docker container commands
- Docker networking and volumes
- Docker Desktop Dashboard

Technical requirements

In this chapter, we will continue to use our local Docker installation. The screenshots in this chapter will be from my preferred operating system, macOS, but the Docker commands we will be running will work on all three of the operating systems on which we have installed Docker so far; however, some of the supporting commands, which will be few and far between, may only be applicable to macOS- and Linux-based operating systems.

Check out the following video to see the Code in Action:
`https://bit.ly/3m1Wtk4`

Understanding Docker container commands

Before we dive into the more complex Docker commands, let's review and go into a little more detail on the commands we have used in previous chapters.

The basics

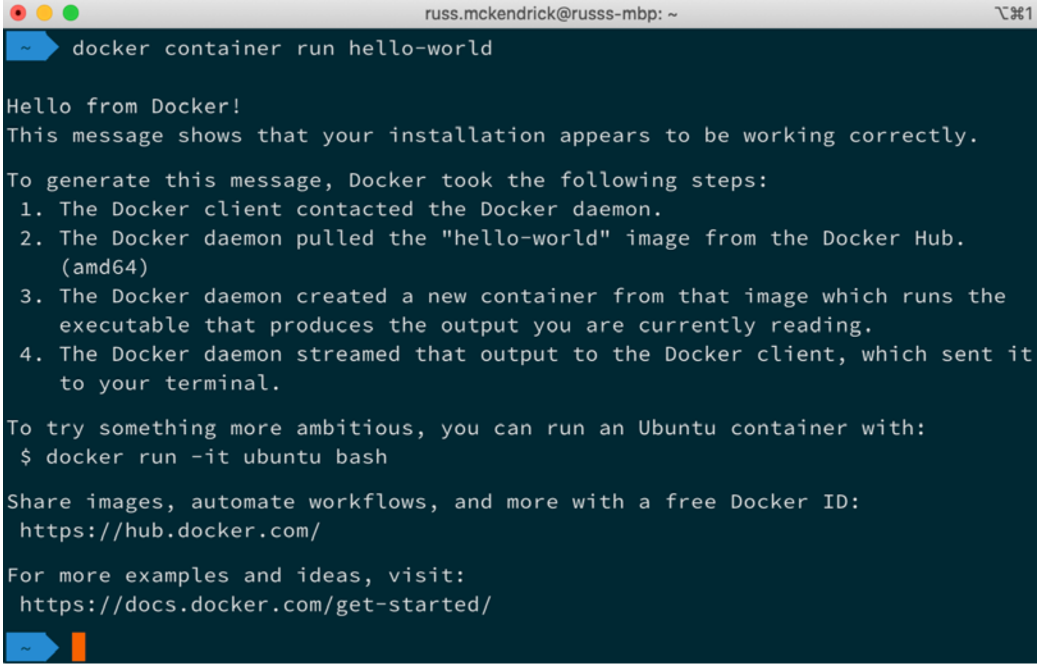
In *Chapter 1*, *Docker Overview*, we launched the most basic container of all, the `hello-world` container, using the following command:

```
$ docker container run hello-world
```

As you may recall, this command pulls a `1.84` KB image from the Docker Hub. You can find the Docker Hub page for the image at `https://hub.docker.com/images/hello-world/`, and, as per the following `Dockerfile`, it runs an executable called `hello`:

```
FROM scratch
COPY hello /
CMD ["/hello"]
```

The `hello` executable prints the `Hello from Docker!` text to the Terminal, and then the process exits. As you can see from the full message text in the following Terminal output, the `hello` binary also lets you know exactly what steps have just occurred:

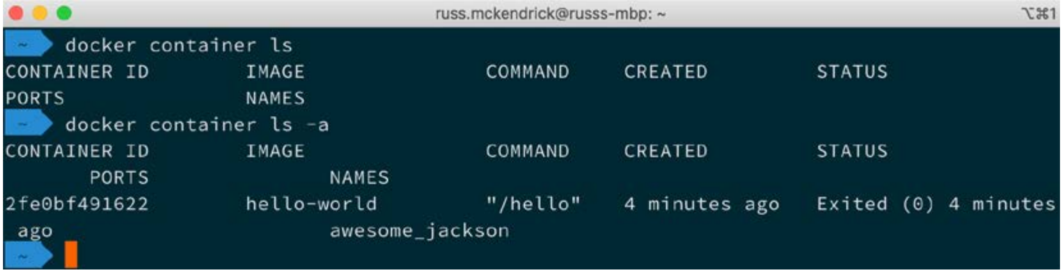


```
~ docker container run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

~
```

Figure 4.1 – Running hello-world

As the process exits, our container also stops. This can be seen by running the following command:

```
$ docker container ls -a
```



```
~ docker container ls
CONTAINER ID        IMAGE               COMMAND     CREATED          STATUS
PORTS               NAMES
~ docker container ls -a
CONTAINER ID        IMAGE               COMMAND     CREATED          STATUS
      PORTS               NAMES
2fe0bf491622        hello-world         "/hello"    4 minutes ago    Exited (0) 4 minutes
 ago                      awesome_jackson
~
```

Figure 4.2 – Listing our containers

You may notice in the Terminal output that I first ran `docker container ls` with and without the `-a` flag. This is shorthand for `--all`, as running it without the flag does not show any exited containers.

You may have noticed that we didn't have to name our container. This is because it wasn't going to be around long enough for us to care what it was called. Docker automatically assigns names for containers, though, and in my case, you can see that it was called `awesome_jackson`.

You will notice throughout your use of Docker that it comes up with some really interesting names for your containers if you choose to let it generate them for you. It created the name from a wordlist for the left-hand word, and for the right-hand word, from the names of notable scientists and hackers. Although this is slightly off topic, the code to generate the names can be found in `names-generator.go`. Right at the end of the source code, it has the following `if` statement:

```
if name == "boring_wozniak" /* Steve Wozniak is not boring */ {
goto begin }
```

This means there will never be a container called boring_wozniak (and quite rightly, too).

> **Information:**
>
> Steve Wozniak is an inventor, electronics engineer, programmer, and entrepreneur who co-founded Apple Inc. with Steve Jobs. He is known as a pioneer of the personal computer revolution of the 70s and 80s, and is definitely not boring!

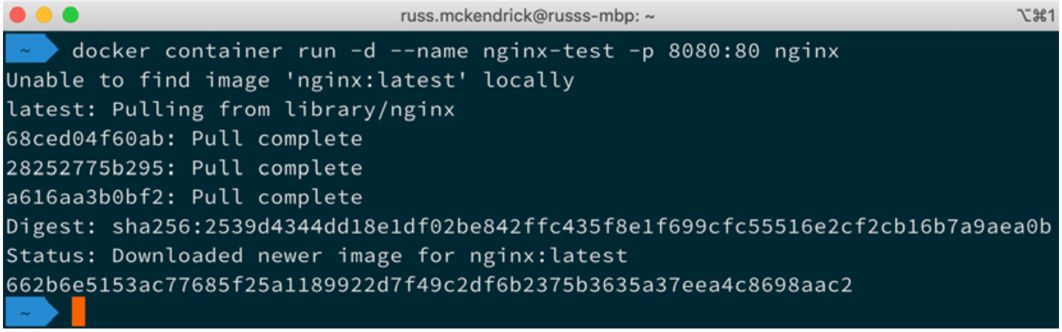
We can remove the container with a status of `exited` by running the following command, making sure that you replace the name of the container with your own container name:

```
$ docker container rm awesome_jackson
```

Also, at the end of *Chapter 1*, *Docker Overview*, we launched a container using the official NGINX image by using the following command:

```
$ docker container run -d --name nginx-test -p 8080:80 nginx
```

As you may remember, this downloads the image and runs it, mapping port `8080` on our host machine to port `80` on the container, and calls it `nginx-test`:



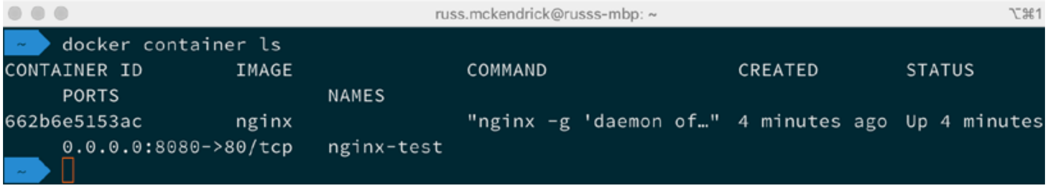
```
~ docker container run -d --name nginx-test -p 8080:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
68ced04f60ab: Pull complete
28252775b295: Pull complete
a616aa3b0bf2: Pull complete
Digest: sha256:2539d4344dd18e1df02be842ffc435f8e1f699cfc55516e2cf2cb16b7a9aea0b
Status: Downloaded newer image for nginx:latest
662b6e5153ac77685f25a1189922d7f49c2df6b2375b3635a37eea4c8698aac2
~
```

Figure 4.3 – Running an NGINX container

As you can see, running the Docker image `ls` shows us that we now have two images downloaded and also running. The following command shows us that we have a running container:

```
$ docker container ls
```

The following Terminal output shows that mine had been up for 5 minutes when I ran the command:



```
~ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED          STATUS
         PORTS                  NAMES
662b6e5153ac        nginx               "nginx -g 'daemon of…"   4 minutes ago    Up 4 minutes
         0.0.0.0:8080->80/tcp   nginx-test
~
```

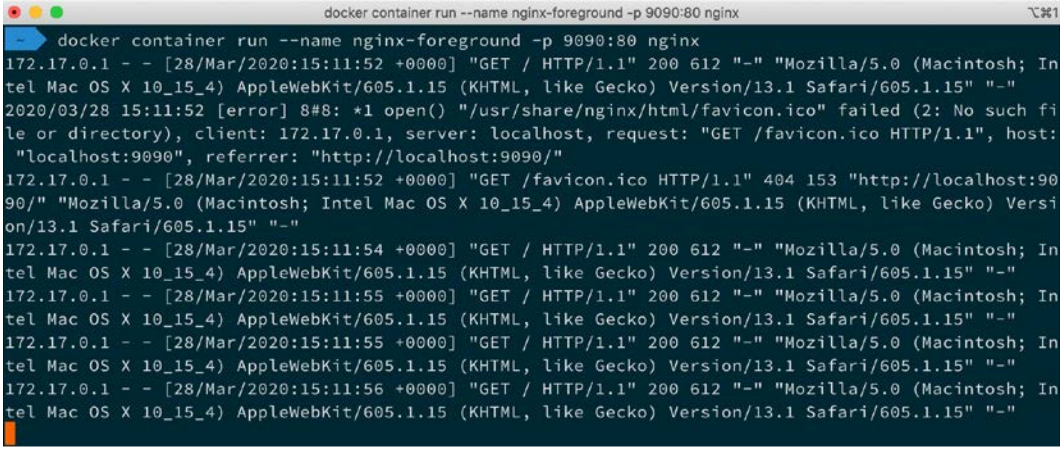
Figure 4.4 – Viewing the running containers

As you can see from our `docker container run` command, we introduced three flags. One of them was `-d`, which is shorthand for `--detach`. If we hadn't added this flag, then our container would have executed in the foreground, which means that our Terminal would have been frozen until we passed the process an escape command by pressing *Ctrl* + *C*.

We can see this in action by running the following command to launch a second NGINX container to run alongside the container we have already launched:

```
$ docker container run --name nginx-foreground -p 9090:80 nginx
```

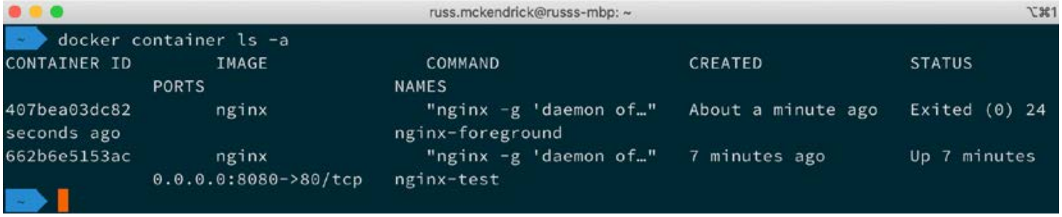
Once launched, open a browser and enter `http://localhost:9090/`. As you load the page, you will notice that your page visit is printed to the screen. Hitting refresh in your browser will display more hits, until you press *Ctrl* + *C* back in the Terminal:


docker container run --name nginx-foreground -p 9090:80 nginx
⌥⌘1

```
~ docker container run --name nginx-foreground -p 9090:80 nginx
172.17.0.1 - - [28/Mar/2020:15:11:52 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
2020/03/28 15:11:52 [error] 8#8: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such fi
le or directory), client: 172.17.0.1, server: localhost, request: "GET /favicon.ico HTTP/1.1", host:
 "localhost:9090", referrer: "http://localhost:9090/"
172.17.0.1 - - [28/Mar/2020:15:11:52 +0000] "GET /favicon.ico HTTP/1.1" 404 153 "http://localhost:90
90/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Versi
on/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:11:54 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:11:55 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:11:55 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:11:56 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
```

Figure 4.5 – Viewing the output of running the container in the foreground

Running `docker container ls -a` shows that you have two containers, one of which has exited:



```
~ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS
             PORTS                  NAMES
407bea03dc82        nginx               "nginx -g 'daemon of…"   About a minute ago   Exited (0) 24
seconds ago                         nginx-foreground
662b6e5153ac        nginx               "nginx -g 'daemon of…"   7 minutes ago        Up 7 minutes
             0.0.0.0:8080->80/tcp   nginx-test
~
```

Figure 4.6 – Listing the running containers

So, what happened? When we removed the `detach` flag, Docker connected us to the NGINX process directly within the container, meaning that we had visibility of **stdin**, **stdout**, and **stderr** for that process. When we used *Ctrl* + *C*, we actually sent an instruction to the NGINX process to terminate it. As that was the process that was keeping our container running, the container exited immediately once there was no longer a running process.

Important note

Standard input (**stdin**) is the handle that our process reads to get information from the end user. **Standard output** (**stdout**) is where the process writes normal information. **Standard error** (**stderr**) is where the process writes error messages.

Another thing you may have noticed when we launched the `nginx-foreground` container is that we gave it a different name using the `--name` flag.

This is because you cannot have two containers with the same name, since Docker gives you the option of interacting with your containers using both the `CONTAINER ID or NAME` values. This is the reason the name generator function exists: to assign a random name to containers you do not wish to name yourself, and also to ensure that we never call Steve Wozniak boring.

The final thing to mention is that when we launched `nginx-foreground`, we asked Docker to map port `9090` to port `80` on the container. This was because we cannot assign more than one process to a port on a host machine, so if we attempted to launch our second container with the same port as the first, we would have received an error message:

```
docker: Error response from daemon: driver failed programming
external connectivity on endpoint nginx-foreground
(3f5b355607f24e03f09a60ee688645f223bafe4492f807459e4a
2b83571f23f4): Bind for 0.0.0.0:8080 failed: port is already
allocated.
```

Also, since we are running the container in the foreground, you may receive an error from the NGINX process, as it failed to start:

```
ERRO[0003] error getting events from daemon: net/http: request
cancelled
```

However, you may also notice that we are mapping to port `80` on the container – why no error there?

Well, as explained in *Chapter 1*, *Docker Overview*, the containers themselves are isolated resources, which means that we can launch as many containers as we like with port `80` remapped, and they will never clash with other containers; we only run into problems when we want to route to the exposed container port from our Docker host.

Let's keep our NGINX container running for the next section, where we will explore more ways of interacting with the container.

Interacting with your containers

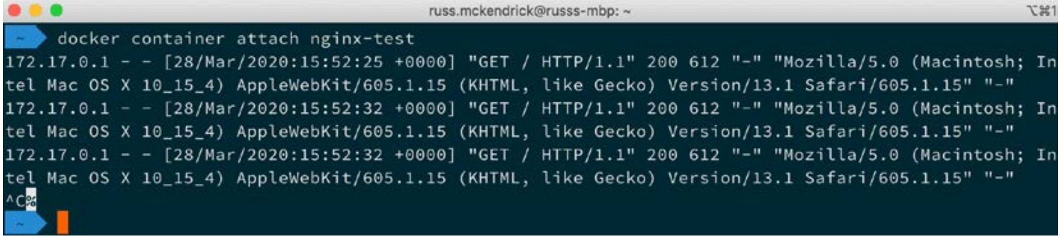
So far, our containers have been running a single process. Docker provides you with a few tools that enable you to both fork additional processes and interact with them, and we will be covering those tools in the following sections:

attach

The first way of interacting with your running container is to attach to the running process. We still have our `nginx-test` container running, so let's connect to that by running this command:

```
$ docker container attach nginx-test
```

Opening your browser and going to `http://localhost:8080/` will print the NGINX access logs to the screen, just like when we launched the `nginx-foreground` container. Pressing *Ctrl* + *C* will terminate the process and return your Terminal to normal. However, as before, we would have terminated the process that was keeping the container running:



```
~ docker container attach nginx-test
172.17.0.1 - - [28/Mar/2020:15:52:25 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:52:32 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:52:32 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
^C%
~
```

Figure 4.7 – Attaching to our container

We can start our container back up by running the following command:

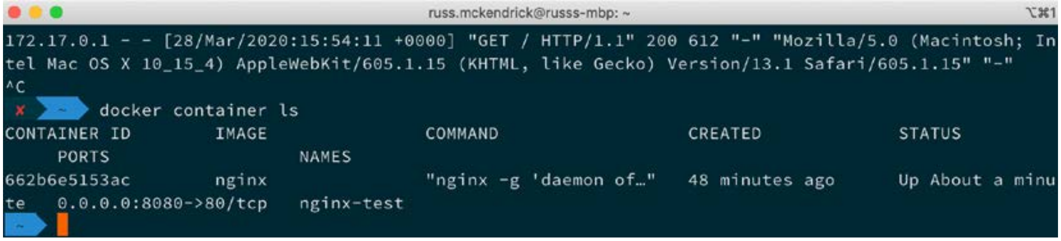
```
$ docker container start nginx-test
```

This will start the container back up in the detached state, meaning that it is running in the background again, as this was the state that the container was originally launched in. Going to `http://localhost:8080/` will show you the NGINX welcome page again.

Let's reattach to our process, but this time with an additional option:

```
$ docker container attach --sig-proxy=false nginx-test
```

Hitting the container's URL a few times and then pressing *Ctrl* + *C* will detach us from the NGINX process, but this time, rather than terminating the NGINX process, it will just return us to our Terminal, leaving the container in a detached state that can be seen by running docker container ls:



```
172.17.0.1 - - [28/Mar/2020:15:54:11 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
^C
 x  ~  docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
    PORTS                  NAMES
662b6e5153ac        nginx               "nginx -g 'daemon of…"   48 minutes ago      Up About a minu
te   0.0.0.0:8080->80/tcp   nginx-test
 ~
```

Figure 4.8 – Disconnecting from our container

This is a great way of quickly attaching to a running container to debug issues while keeping the container's main process up and running.

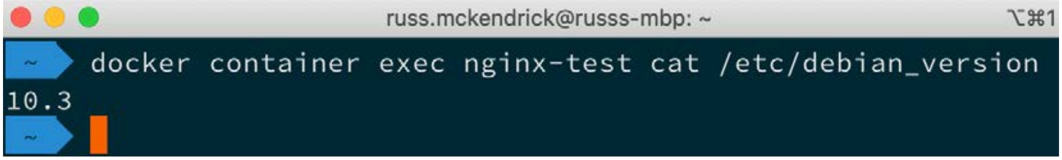
exec

The `attach` command is useful if you need to connect to the process your container is running, but what if you need something that is a little more interactive?

You can use the `exec` command. This spawns a second process within the container that you can interact with. For example, to see the contents of the `/etc/debian_version` file, we can run the following command:

```
$ docker container exec nginx-test cat /etc/debian_version
```

This will spawn a second process, the `cat` command in this case, which prints the contents of `/etc/debianversion to stdout`. The second process will then terminate, leaving our container as it was before the `exec` command was executed:



```
 ~  docker container exec nginx-test cat /etc/debian_version
10.3
 ~
```

Figure 4.9 – Executing a command against our container

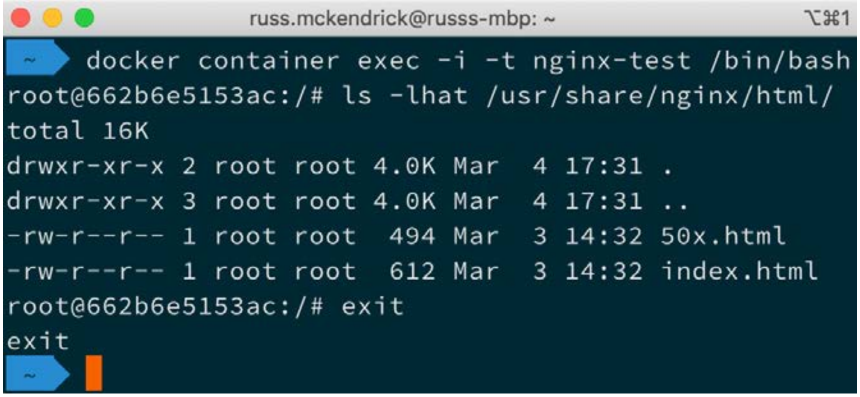
We can take this one step further by running the following command:

```
$ docker container exec -i -t nginx-test /bin/bash
```

This time, we are forking a bash process and using the `-i` and `-t` flags to keep open console access to our container. The `-i` flag is shorthand for `--interactive`, which instructs Docker to keep `stdin` open so that we can send commands to the process. The `-t` flag is short for `–tty` and allocates a pseudo-TTY to the session.

> **Important note**
>
> Early user terminals connected to computers were called **teletypewriters**. While these devices are no longer used today, the acronym TTY has continued to be used to described text-only consoles in modern computing.

What this means is that you will be able to interact with the container as if you had a remote Terminal session, like SSH:



```
~ docker container exec -i -t nginx-test /bin/bash
root@662b6e5153ac:/# ls -lhat /usr/share/nginx/html/
total 16K
drwxr-xr-x 2 root root 4.0K Mar  4 17:31 .
drwxr-xr-x 3 root root 4.0K Mar  4 17:31 ..
-rw-r--r-- 1 root root  494 Mar  3 14:32 50x.html
-rw-r--r-- 1 root root  612 Mar  3 14:32 index.html
root@662b6e5153ac:/# exit
exit
~
```

Figure 4.10 – Opening an interactive session to our container

While this is extremely useful, as you can interact with the container as if it were a virtual machine, I do not recommend making any changes to your containers as they are running using the pseudo-TTY. It is more than likely that those changes will not persist and will be lost when your container is removed. We will go into the thinking behind this in more detail in *Chapter 15*, *Docker Workflows*.

Now that we have covered the various methods you can connect to and interact with your containers, we are going to look at some of the tools provided by Docker that mean you shouldn't have to.

Logs and process information

So far, we have been attaching to either the process in our container, or to the container itself, in order to view information. Docker provides the commands that we are going to cover in this section that allow you to view information about your containers without having to use either the `attach` or `exec` commands.

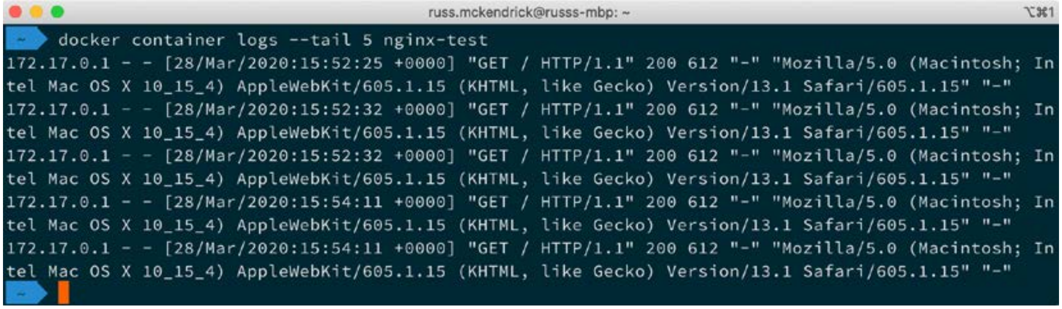
Let's start by looking at how you c view the output being generated by the process within the container without having to run it in the foreground.

logs

The `logs` command is pretty self-explanatory. It allows you to interact with the `stdout` stream of your containers, which Docker is keeping track of in the background. For example, to view the last entries written to `stdout` for our `nginx-test` container, you just need to use the following command:

```
$ docker container logs --tail 5 nginx-test
```

The output of the command is shown here:



```
~ docker container logs --tail 5 nginx-test
172.17.0.1 - - [28/Mar/2020:15:52:25 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:52:32 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:52:32 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:54:11 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:54:11 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
~
```

Figure 4.11 – Tailing the logs

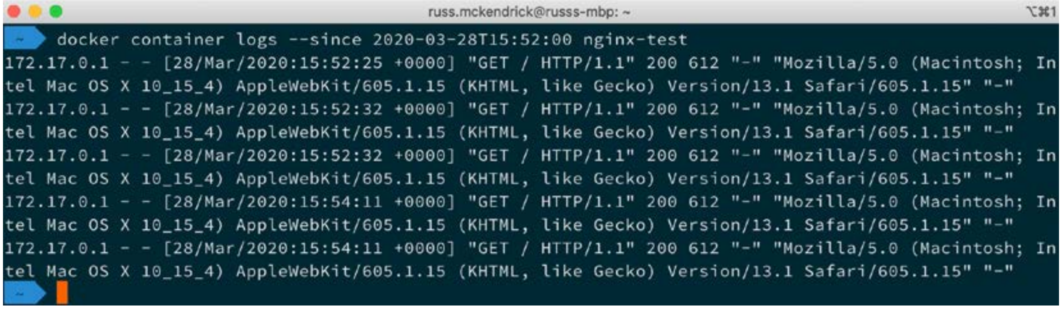
To view the logs in real time, I simply need to run the following:

```
$ docker container logs -f nginx-test
```

The `-f` flag is shorthand for `--follow`. I can also, for example, view everything that has been logged since a certain time by running the following command:

```
$ docker container logs --since 2020-03-28T15:52:00 nginx-test
```

The output of the command is shown here:



```
~ docker container logs --since 2020-03-28T15:52:00 nginx-test
172.17.0.1 - - [28/Mar/2020:15:52:25 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:52:32 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:52:32 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:54:11 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
172.17.0.1 - - [28/Mar/2020:15:54:11 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1 Safari/605.1.15" "-"
~
```

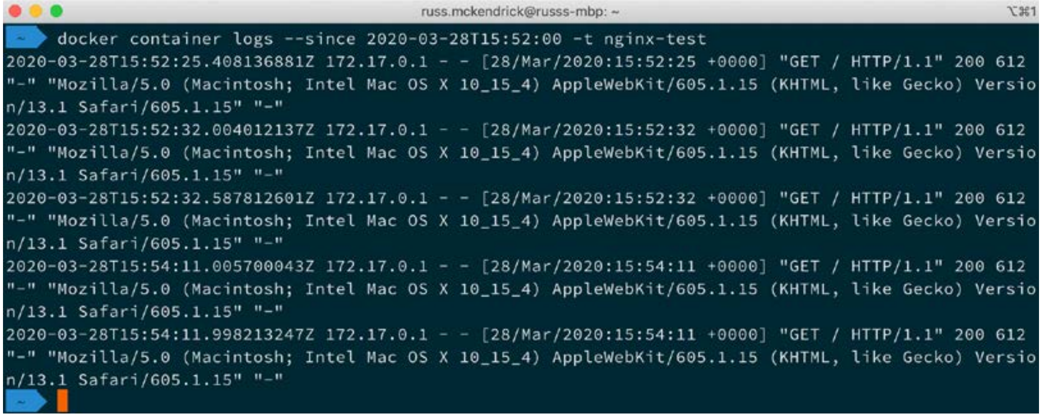
Figure 4.12 – Checking the logs after a certain time

If you notice that the timestamp in the access log is different to that which you are searching for, that is due to the logs command that shows the timestamps of stdout as recorded by Docker, and not the time within the container. An example of this would be the hours' time difference between the host machine and the container due to **British Summer Time (BST)**.

Luckily, to save confusion, you can add `-t` to your `logs` command:

```
$ docker container logs --since 2020-03-28T15:52:00 -t nginx-
test
```

The `-t` flag is short for `--timestamp`; this option prepends the time the output was captured by Docker:



```
~ docker container logs --since 2020-03-28T15:52:00 -t nginx-test
2020-03-28T15:52:25.408136881Z 172.17.0.1 - - [28/Mar/2020:15:52:25 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/13.1 Safari/605.1.15" "-"
2020-03-28T15:52:32.004012137Z 172.17.0.1 - - [28/Mar/2020:15:52:32 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/13.1 Safari/605.1.15" "-"
2020-03-28T15:52:32.587812601Z 172.17.0.1 - - [28/Mar/2020:15:52:32 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/13.1 Safari/605.1.15" "-"
2020-03-28T15:54:11.005700043Z 172.17.0.1 - - [28/Mar/2020:15:54:11 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/13.1 Safari/605.1.15" "-"
2020-03-28T15:54:11.998213247Z 172.17.0.1 - - [28/Mar/2020:15:54:11 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/13.1 Safari/605.1.15" "-"
~
```

Figure 4.13 – Viewing the logs and with the time the entry was logged

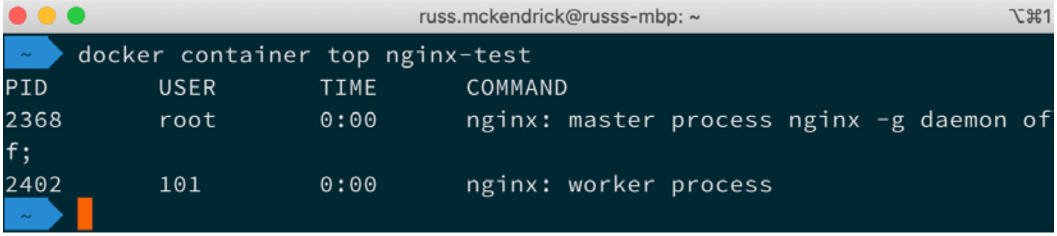
Now what we have looked at the ways we can view the output of the processes running in our containers, let's examine how we get more detail on the process itself.

top

The `top` command is quite a simple one; it lists the processes running within the container that you specify, and is used as follows:

```
$ docker container top nginx-test
```

The output of the command is shown here:



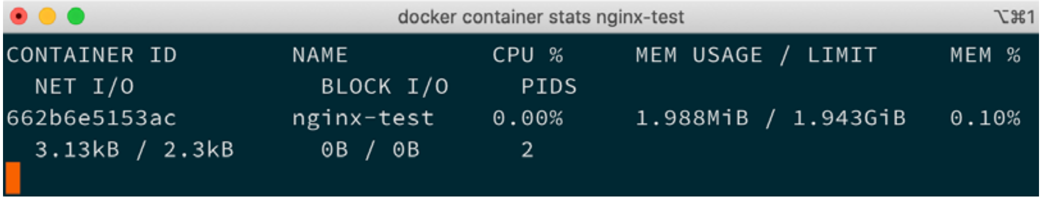
```
~ docker container top nginx-test
PID        USER       TIME       COMMAND
2368       root       0:00       nginx: master process nginx -g daemon of
f;
2402       101        0:00       nginx: worker process
~
```

Figure 4.14 – Running the top command

As you can see from the following Terminal output, we have two processes running, both of which are NGINX, which is to be expected.

stats

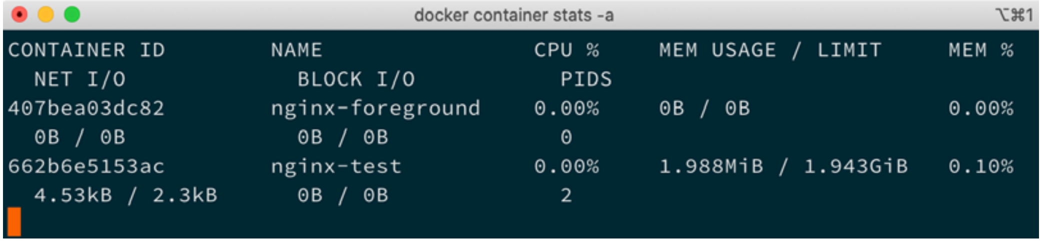
The `stats` command provides real-time information on either the specified container or, if you don't pass a `NAME` or `ID` container, on all running containers:



```
CONTAINER ID        NAME          CPU %       MEM USAGE / LIMIT       MEM %
  NET I/O             BLOCK I/O     PIDS
662b6e5153ac        nginx-test    0.00%       1.988MiB / 1.943GiB     0.10%
  3.13kB / 2.3kB      0B / 0B       2
```

Figure 4.15 – Viewing the real-time stats of a single container

As you can see from the following Terminal output, we are given information on `CPU`, `RAM`, `NETWORK`, `DISK IO`, and `PIDS` for the specified container:



```
CONTAINER ID        NAME                CPU %       MEM USAGE / LIMIT       MEM %
  NET I/O             BLOCK I/O           PIDS
407bea03dc82        nginx-foreground    0.00%       0B / 0B                 0.00%
  0B / 0B             0B / 0B             0
662b6e5153ac        nginx-test          0.00%       1.988MiB / 1.943GiB     0.10%
  4.53kB / 2.3kB      0B / 0B             2
```

Figure 4.16 – Viewing the real-time stats of all running containers

However, as you can see from the preceding output, if the container isn't running, there aren't any resources being utilized, so it doesn't really add any value, other than giving you a visual representation of how many containers you have running and where the resources are being used.

It is also worth pointing out that the information displayed by the `stats` command is real time only; Docker does not record the resource utilization and make it available in the same way that the `logs` command does. We will be looking at more long-term storage options for resource utilization in later chapters.

Resource limits

The `stats` command we ran showed us the resource utilization of our containers. By default, when launched, a container will be allowed to consume all the available resources on the host machine if it so requires. We can put limits on the resources our containers can consume. Let's start by updating the resource allowances of our nginx-test container.

Typically, we would have set the limits when we launched our container using the `run` command; for example, to halve the CPU priority and set a memory limit of `128M`, we would have used the following command:

```
$ docker container run -d --name nginx-test --cpu-shares 512
--memory 128M -p 8080:80 nginx
```

However, we didn't launch our nginx-test container with any resource limits, meaning that we need to update our already running container. To do this, we can use the `update` command. Now, you may have thought that this should just entail running the following command:

```
$ docker container update --cpu-shares 512 --memory 128M nginx-
test
```

But actually, running the preceding command will produce an error:

```
Error response from daemon: Cannot update container
662b6e5153ac77685f25a1189922d7f49c2df6b2375b3635a37eea
4c8698aac2: Memory limit should be smaller than already set
memoryswap limit, update the memoryswap at the same time
```

So, what is the `memoryswap` limit currently set to? To find this out, we can use the `inspect` command to display all of the configuration data for our running container; just run the following command:

```
$ docker container inspect nginx-test
```

If you are following along, then you will see that running the preceding command, there is a lot of configuration data that will be displayed, too much to display here. When I ran the command, a `199 line JSON` array was returned. Let's use the `grep` command to filter out just the lines that contain the word `memory`:

```
$ docker container inspect nginx-test | grep -i memory
```

This returns the following configuration data:

```
            "Memory": 0,
            "KernelMemory": 0,
            "KernelMemoryTCP": 0,
            "MemoryReservation": 0,
            "MemorySwap": 0,
            "MemorySwappiness": null,
```

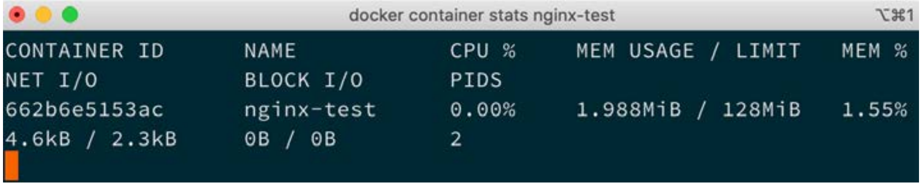
Everything is set to `0`, so how can `128M` be smaller than 0?

```
$ docker container update --cpu-shares 512 --memory 128M
--memory-swap 256M nginx-test
```

In the context of the configuration of the resources, `0` is actually the default value and means that there are no limits. Notice the lack of `M` after each numerical value. This means that our `update` command should actually read as the preceding command.

> **Important note**
>
> **Paging** is a memory management scheme in which the kernel stores and retrieves, or swaps, data from secondary storage for use in the main memory. This allows processes to exceed the size of physical memory available.

By default, when you set `--memory` as part of the `run` command, Docker will set `--memory-swap size` to be twice that of `--memory`. If you run `docker container stats nginx-test` now, you should see our limits in place:



```
CONTAINER ID        NAME                CPU %        MEM USAGE / LIMIT     MEM %
NET I/O             BLOCK I/O           PIDS
662b6e5153ac        nginx-test          0.00%        1.988MiB / 128MiB     1.55%
4.6kB / 2.3kB       0B / 0B             2
```

Figure 4.17 – Using stats to view the limits

Also, re-running `docker container inspect nginx-test | grep -i memory` will show the changes as follows:

```
                "Memory": 134217728,
                "KernelMemory": 0,
                "KernelMemoryTCP": 0,
                "MemoryReservation": 0,
                "MemorySwap": 268435456,
                "MemorySwappiness": null,
```

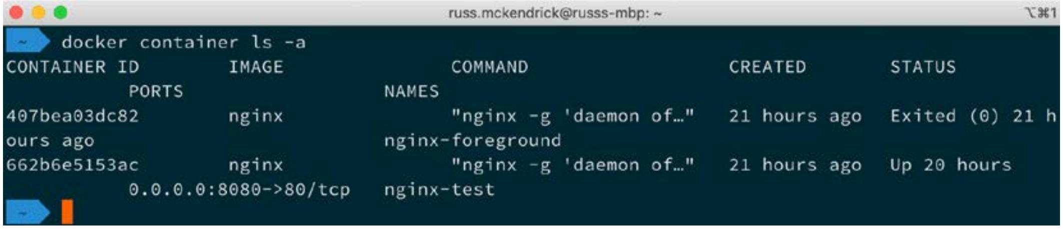
You will notice that while we defined the values in MB, they are displayed here in bytes, so they are correct.

> **Tip**
>
> The values when running `docker container inspect` are all shown in bytes rather than megabytes (MB).

Container states and miscellaneous commands

For the final part of this section, we are going to look at the various states your containers could be in, along with the few remaining commands we have yet to cover as part of the `docker container` command.

Running `docker container ls -a` should show something similar to the following Terminal output:



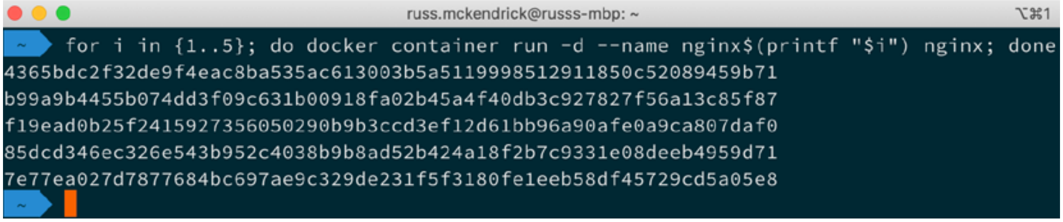
```
~ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED          STATUS
         PORTS                NAMES
407bea03dc82        nginx               "nginx -g 'daemon of…"   21 hours ago     Exited (0) 21 h
ours ago                      nginx-foreground
662b6e5153ac        nginx               "nginx -g 'daemon of…"   21 hours ago     Up 20 hours
         0.0.0.0:8080->80/tcp   nginx-test
~
```

Figure 4.18 – Listing all of the containers, including those that have exited

As you can see, we have two containers; one has the status of `Up` and the other has `Exited`. Before we continue, let's launch five more containers. To do this quickly, run the following command:

```
$ for i in {1..5}; do docker container run -d --name
nginx$(printf "$i") nginx; done
```

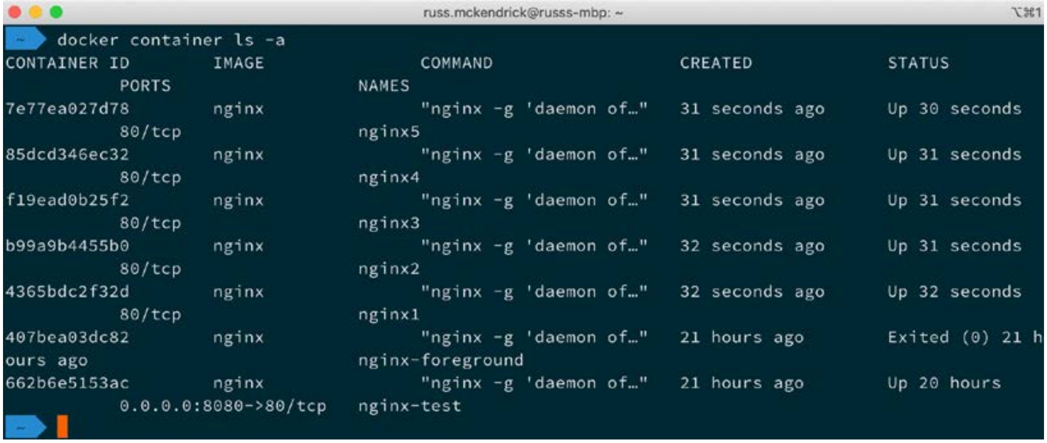
You should see something like the following output:



```
~ for i in {1..5}; do docker container run -d --name nginx$(printf "$i") nginx; done
4365bdc2f32de9f4eac8ba535ac613003b5a5119998512911850c52089459b71
b99a9b4455b074dd3f09c631b00918fa02b45a4f40db3c927827f56a13c85f87
f19ead0b25f2415927356050290b9b3ccd3ef12d61bb96a90afe0a9ca807daf0
85dcd346ec326e543b952c4038b9b8ad52b424a18f2b7c9331e08deeb4959d71
7e77ea027d7877684bc697ae9c329de231f5f3180fe1eeb58df45729cd5a05e8
~
```

Figure 4.19 – Launching five containers quickly

When running `docker container ls -a`, you should see your five new containers, named `nginx1` through to `nginx5`:



```
~ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
           PORTS                  NAMES
7e77ea027d78        nginx               "nginx -g 'daemon of…"   31 seconds ago      Up 30 seconds
           80/tcp                 nginx5
85dcd346ec32        nginx               "nginx -g 'daemon of…"   31 seconds ago      Up 31 seconds
           80/tcp                 nginx4
f19ead0b25f2        nginx               "nginx -g 'daemon of…"   31 seconds ago      Up 31 seconds
           80/tcp                 nginx3
b99a9b4455b0        nginx               "nginx -g 'daemon of…"   32 seconds ago      Up 31 seconds
           80/tcp                 nginx2
4365bdc2f32d        nginx               "nginx -g 'daemon of…"   32 seconds ago      Up 32 seconds
           80/tcp                 nginx1
407bea03dc82        nginx               "nginx -g 'daemon of…"   21 hours ago        Exited (0) 21 h
ours ago                                  nginx-foreground
662b6e5153ac        nginx               "nginx -g 'daemon of…"   21 hours ago        Up 20 hours
           0.0.0.0:8080->80/tcp   nginx-test
~ 
```

Figure 4.20 – Viewing our five new containers

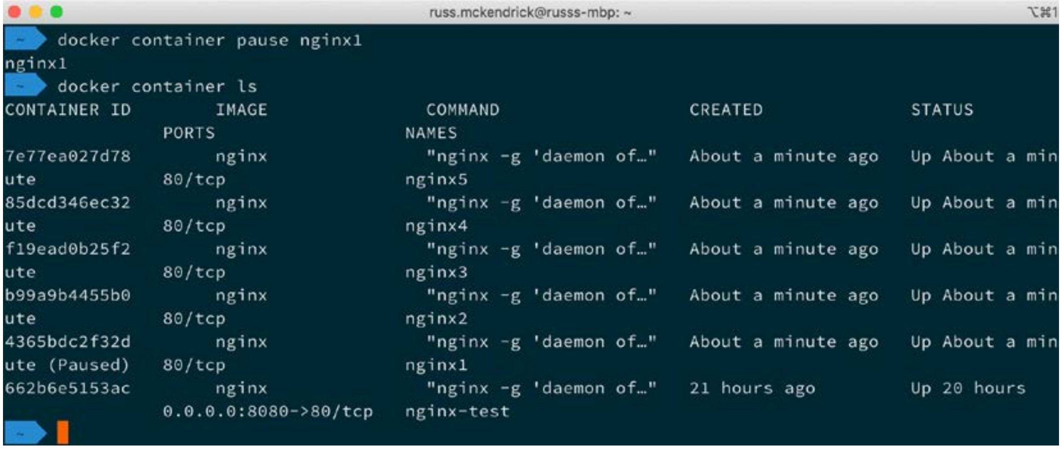
Now that we have the additional containers up and running, let's look at how we can their states.

pause and unpause

Let's look at pausing `nginx1`. To do this, simply run the following:

```
$ docker container pause nginx1
```

Running `docker container ls` will show that the container has a status of `Up`, but it also says `Paused`:



```
~  docker container pause nginx1
nginx1
~  docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS
          PORTS                  NAMES
7e77ea027d78        nginx               "nginx -g 'daemon of…"   About a minute ago   Up About a min
ute                 80/tcp               nginx5
85dcd346ec32        nginx               "nginx -g 'daemon of…"   About a minute ago   Up About a min
ute                 80/tcp               nginx4
f19ead0b25f2        nginx               "nginx -g 'daemon of…"   About a minute ago   Up About a min
ute                 80/tcp               nginx3
b99a9b4455b0        nginx               "nginx -g 'daemon of…"   About a minute ago   Up About a min
ute                 80/tcp               nginx2
4365bdc2f32d        nginx               "nginx -g 'daemon of…"   About a minute ago   Up About a min
ute (Paused)        80/tcp               nginx1
662b6e5153ac        nginx               "nginx -g 'daemon of…"   21 hours ago         Up 20 hours
                    0.0.0.0:8080->80/tcp nginx-test
~  
```

Figure 4.21 – Pausing a container

Note that we didn't have to use the `-a` flag to see information about the container as the process has not been terminated; instead, it has been suspended using the `cgroups` freezer. With the `cgroups` freezer, the process is unaware it has been suspended, meaning that it can be resumed.

As you will have probably already guessed, you can resume a paused container using the `unpause` command, as follows:

```
$ docker container unpause nginx1
```

This command is useful if you need to freeze the state of a container; for example, maybe one of your containers is going haywire and you need to do some investigation later, but don't want it to have a negative impact on your other running containers.

Now, let's look at how you can properly stop and remove containers.

stop, start, restart, and kill

Next up, we have the `stop`, `start`, `restart`, and `kill` commands. We have already used the `start` command to resume a container with a status of `Exited`. The `stop` command works exactly the same way as when we used *Ctrl* + *C* to detach from your container running in the foreground.

Run the following command:

```
$ docker container stop nginx2
```

With this, a request is sent to the process for it to terminate, called `SIGTERM`. If the process has not terminated itself within a grace period, then a kill signal, called `SIGKILL`, is sent. This will immediately terminate the process, not giving it any time to finish whatever is causing the delay; for example, committing the results of a database query to disk.

Because this could be bad, Docker gives you the option of overriding the default grace period, which is 10 seconds, by using the `-t` flag; this is short for `--time`. For example, running the following command will wait up to 60 seconds before sending a `SIGKILL` command, in the event that it needs to be sent to kill the process:

```
$ docker container stop -t 60 nginx3
```

The `start` command, as we have already seen, will start the process back up; however, unlike the `pause` and `unpause` commands, the process, in this case, starts from scratch using the flags that originally launched it, rather than starting from where it left off:

```
$ docker container start nginx2 nginx3
```

The `restart` command is a combination of the following two commands; it stops and then starts the `ID` or `NAME` container you pass it. Also, as with `stop`, you can pass the `-t` flag:

```
$ docker container restart -t 60 nginx4
```

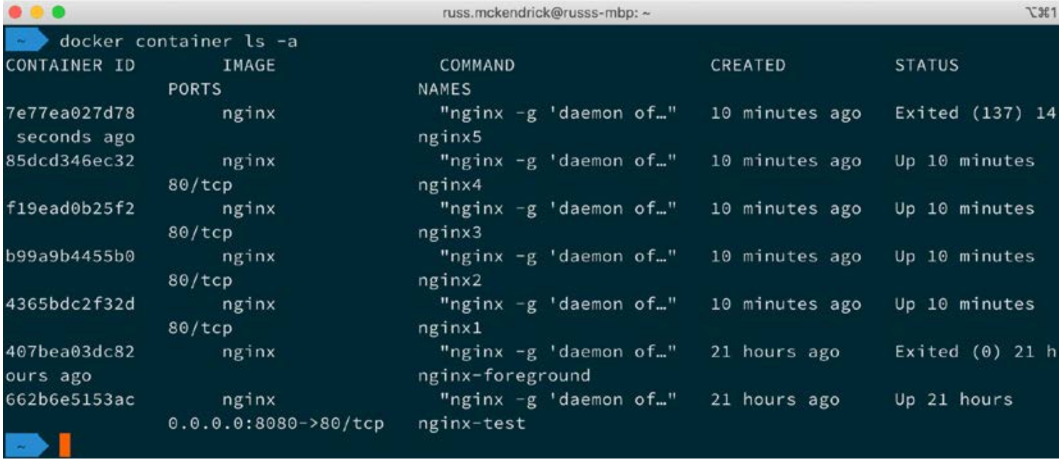
Finally, you also have the option of sending a `SIGKILL` command immediately to the container by running the `kill` command:

```
$ docker container kill nginx5
```

There is one more thing need to cover, and that is removing the containers.

Removing containers

Let's check the status of containers we have been using by the `docker container ls -a` command. When I run the command, I can see that I have two containers with an `Exited` status and all of the others are running:



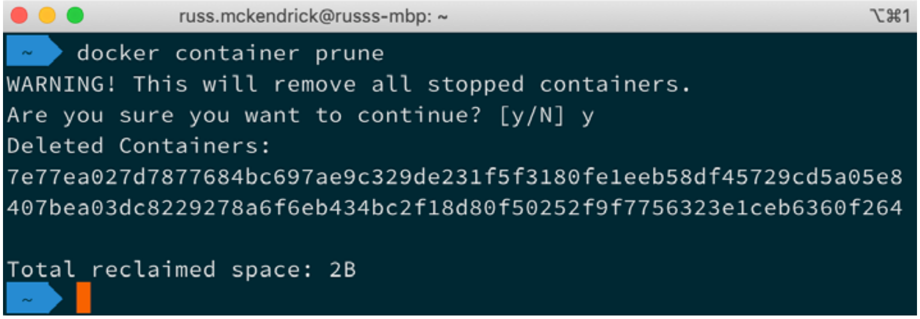
```
russ.mckendrick@russs-mbp: ~                                              ⌥⌘1
 ~  docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
          PORTS                  NAMES
7e77ea027d78        nginx               "nginx -g 'daemon of…"   10 minutes ago      Exited (137) 14
 seconds ago                            nginx5
85dcd346ec32        nginx               "nginx -g 'daemon of…"   10 minutes ago      Up 10 minutes
          80/tcp                        nginx4
f19ead0b25f2        nginx               "nginx -g 'daemon of…"   10 minutes ago      Up 10 minutes
          80/tcp                        nginx3
b99a9b4455b0        nginx               "nginx -g 'daemon of…"   10 minutes ago      Up 10 minutes
          80/tcp                        nginx2
4365bdc2f32d        nginx               "nginx -g 'daemon of…"   10 minutes ago      Up 10 minutes
          80/tcp                        nginx1
407bea03dc82        nginx               "nginx -g 'daemon of…"   21 hours ago        Exited (0) 21 h
ours ago                                nginx-foreground
662b6e5153ac        nginx               "nginx -g 'daemon of…"   21 hours ago        Up 21 hours
          0.0.0.0:8080->80/tcp          nginx-test
 ~ 
```

Figure 4.22 – Viewing the state of all of our containers

To remove the two exited containers, I can simply run the `prune` command:

```
$ docker container prune
```

When doing so, a warning pops up asking you to confirm whether you are really sure, as seen in the following screenshot:



```
russ.mckendrick@russs-mbp: ~                                  ⌥⌘1
 ~  docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
7e77ea027d7877684bc697ae9c329de231f5f3180fe1eeb58df45729cd5a05e8
407bea03dc8229278a6f6eb434bc2f18d80f50252f9f7756323e1ceb6360f264

Total reclaimed space: 2B
 ~ 
```

Figure 4.23 – Pruning the containers

You can choose which container you want to remove using the `rm` command, an example of which is shown here:

```
$ docker container rm nginx4
```

Another alternative would be to string the `stop` and `rm` commands together:

```
$ docker container stop nginx3 && docker container rm nginx3
```

However, given that you can use the `prune` command now, this is probably way too much effort, especially as you are trying to remove the containers and probably don't care too much how gracefully the process is terminated.

Feel free to remove the remainder of your containers using whichever method you like.

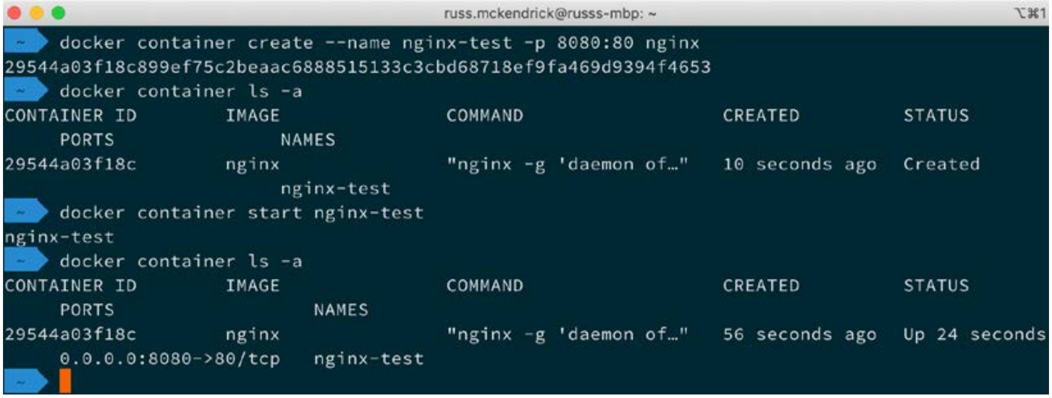
Before we wrap up this section of the chapter, we are going to look at a few more useful commands that can't be really grouped together.

Miscellaneous commands

For the final part of this section, we are going to look at a few commands that you probably won't use too much during your day-to-day use of Docker. The first of these is `create`. The `create` command is pretty similar to the `run` command, except that it does not start the container, but instead prepares and configures one:

```
$ docker container create --name nginx-test -p 8080:80 nginx
```

You can check the status of your created container by running `docker container ls -a`, and then starting the container with `docker container start nginx-test`, before checking the status again:



```
 ~  docker container create --name nginx-test -p 8080:80 nginx
29544a03f18c899ef75c2beaac6888515133c3cbd68718ef9fa469d9394f4653
 ~  docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED          STATUS
    PORTS               NAMES
29544a03f18c        nginx               "nginx -g 'daemon of…"   10 seconds ago   Created
                        nginx-test
 ~  docker container start nginx-test
nginx-test
 ~  docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED          STATUS
    PORTS                  NAMES
29544a03f18c        nginx               "nginx -g 'daemon of…"   56 seconds ago   Up 24 seconds
    0.0.0.0:8080->80/tcp   nginx-test
 ~ 
```

Figure 4.24 – Creating and then running a container

The next command we are going to quickly look at is the `port` command; this displays the `port` number along with any port mappings for the container:

```
$ docker container port nginx-test
```

It should return the following:

```
     80/tcp -> 0.0.0.0:8080
```

We already know this, as it is what we configured. Also, the ports are listed in the `docker container ls` output.

The next command we are going to look at quickly is the `diff` command. This command prints a list of all of the files that have been added (`A`) or changed (`C`) since the container was started – so basically, a list of the differences in the filesystem between the original image we used to launch the container and what files are present now.

Before we run the command, let's create a blank file within the `nginx-test` container using the `exec` command:

```
$ docker container exec nginx-test touch /tmp/testing
```

Now that we have a file called `testing` in `/tmp`, we can view the differences between the original image and the running container using the following command:

```
$ docker container diff nginx-test
```

This will return a list of files. As you can see from the following list, our testing file is there, along with the files that were created when NGINX started:

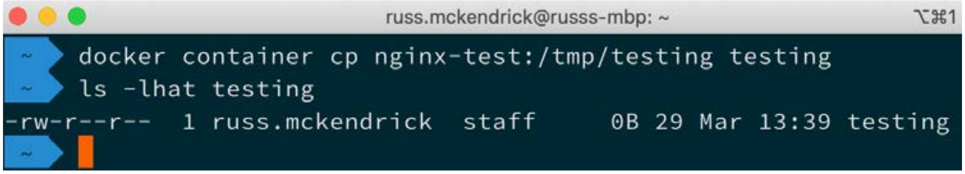
```
     C /run
     A /run/nginx.pid
     C /tmp
     A /tmp/testing
     C /var
     C /var/cache
     C /var/cache/nginx
     A /var/cache/nginx/client_temp
     A /var/cache/nginx/fastcgi_temp
     A /var/cache/nginx/proxy_temp
     A /var/cache/nginx/scgi_temp
     A /var/cache/nginx/uwsgi_temp
```

It is worth pointing out that once we stop and remove the container, these files will be lost. In the next section of this chapter, we will look at Docker volumes and learn how we can persist data. Before we move on though, let's get a copy of the file we just created using the `cp` command.

To do this, we can run the following:

```
$ docker container cp nginx-test:/tmp/testing testing
```

As you can see from the command, we are providing the container name followed by `:` and the full path to the file we want to copy. What follows is the local path. Here, you can see that we are simply calling the file `testing` and it will be copied to the current folder:



```
~ docker container cp nginx-test:/tmp/testing testing
~ ls -lhat testing
-rw-r--r--  1 russ.mckendrick  staff     0B 29 Mar 13:39 testing
~
```

Figure 4.25 – Copying a file to a container

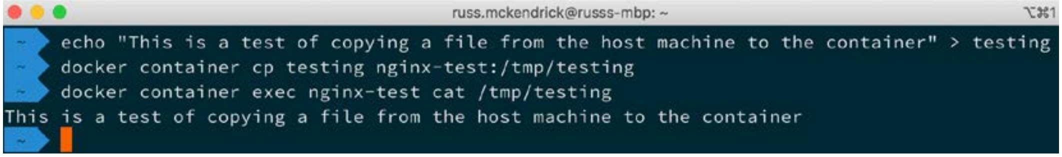
As the file does not contain any data, lets add some and then copy it back to the container:

```
$ echo "This is a test of copying a file from the host machine
to the container" > testing
```

```
$ docker container cp testing nginx-test:/tmp/testing
```

```
$ docker container exec nginx-test cat /tmp/testing
```

Notice that in the second command, we are swapping the paths around. This time, we are providing the path of the local file and the container name and path:



```
~ echo "This is a test of copying a file from the host machine to the container" > testing
~ docker container cp testing nginx-test:/tmp/testing
~ docker container exec nginx-test cat /tmp/testing
This is a test of copying a file from the host machine to the container
~
```

Figure 4.26 – Copying a file with contents to the container

The other thing of note is that while we are overwriting an existing file, Docker did not warn us or give an option to back out of the command – it went ahead and overwrote the file immediately, so please be careful when using `docker container cp`.

If you are following along, you should remove any running containers launched during this section using the command of your choice before moving on.

Docker networking and volumes

Next up, we are going to take a look at the basics of Docker networking and Docker volumes using the default drivers. Let's take a look at networking first.

Docker networking

So far, we have been launching our containers on a single flat shared network. Although we have not talked about it yet, this means the containers we have been launching would have been able to communicate with each other without having to use any of the host networking.

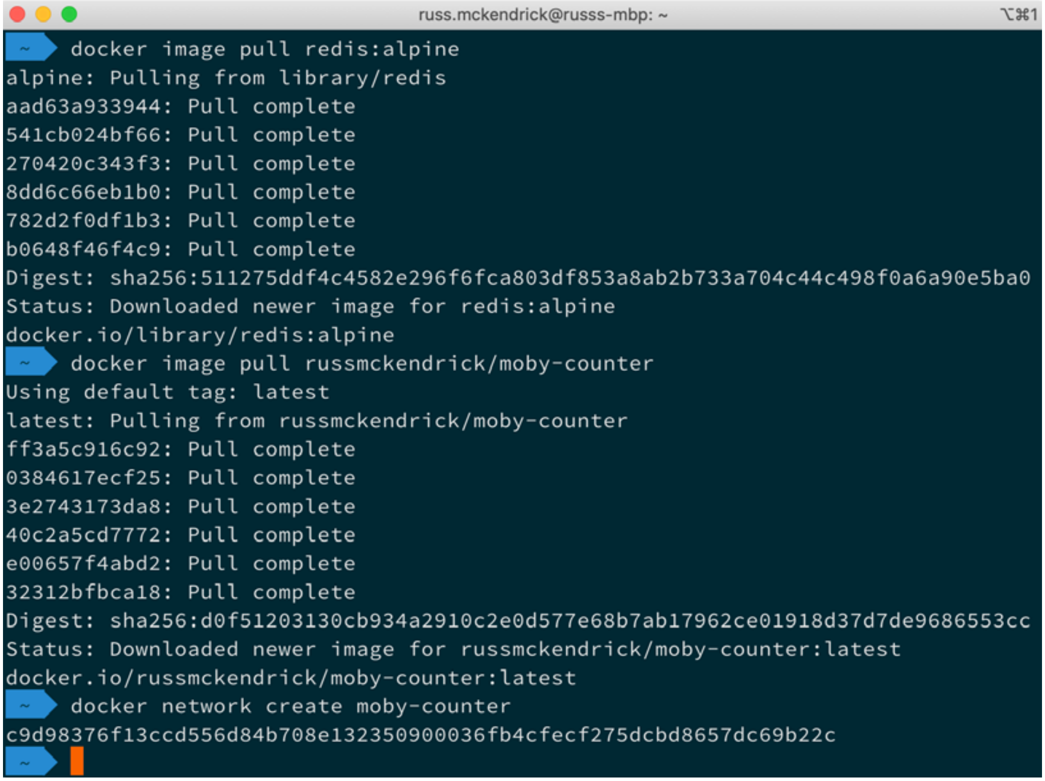
Rather than going into detail now, let's work through an example. We are going to be running a two-container application; the first container will be running Redis, and the second, our application, which uses the Redis container to store a system state.

> **Important note**
>
> Redis is an in-memory data structure store that can be used as a database, cache, or message broker. It supports different levels of on-disk persistence.

Before we launch our application, let's download the container images we will be using, and also create the network:

```
$ docker image pull redis:alpine
$ docker image pull russmckendrick/moby-counter
$ docker network create moby-counter
```

You should see something similar to the following Terminal output:



```
~ docker image pull redis:alpine
alpine: Pulling from library/redis
aad63a933944: Pull complete
541cb024bf66: Pull complete
270420c343f3: Pull complete
8dd6c66eb1b0: Pull complete
782d2f0df1b3: Pull complete
b0648f46f4c9: Pull complete
Digest: sha256:511275ddf4c4582e296f6fca803df853a8ab2b733a704c44c498f0a6a90e5ba0
Status: Downloaded newer image for redis:alpine
docker.io/library/redis:alpine
~ docker image pull russmckendrick/moby-counter
Using default tag: latest
latest: Pulling from russmckendrick/moby-counter
ff3a5c916c92: Pull complete
0384617ecf25: Pull complete
3e2743173da8: Pull complete
40c2a5cd7772: Pull complete
e00657f4abd2: Pull complete
32312bfbca18: Pull complete
Digest: sha256:d0f51203130cb934a2910c2e0d577e68b7ab17962ce01918d37d7de9686553cc
Status: Downloaded newer image for russmckendrick/moby-counter:latest
docker.io/russmckendrick/moby-counter:latest
~ docker network create moby-counter
c9d98376f13ccd556d84b708e132350900036fb4cfecf275dcbd8657dc69b22c
~
```

Figure 4.27 – Pulling the images we need and creating the network

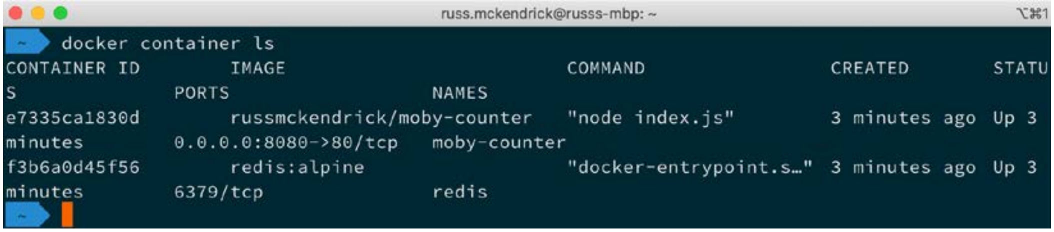
Now that we have our images pulled and our network created, we can launch our containers, starting with the Redis one:

```
$ docker container run -d --name redis --network moby-counter
redis:alpine
```

As you can see, we used the `--network` flag to define the network that our container was launched in. Now that the Redis container is launched, we can launch the application container by running the following command:

```
$ docker container run -d --name moby-counter --network moby-
counter -p 8080:80 russmckendrick/moby-counter
```

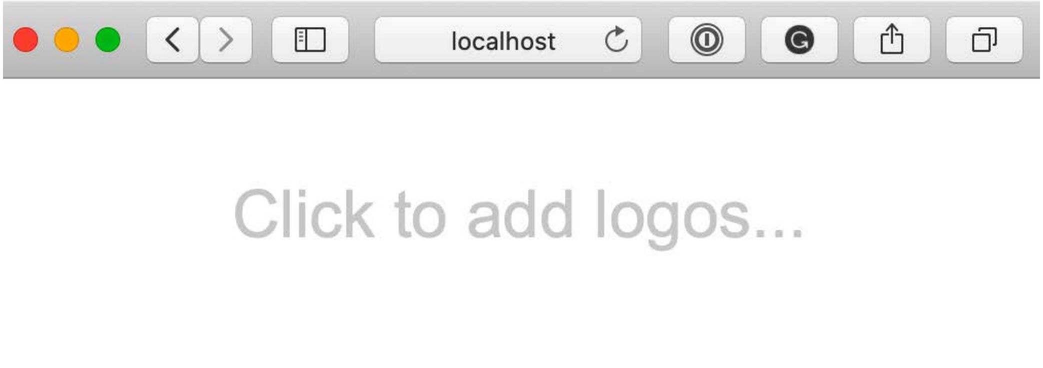
Again, we launched the container on the `moby-counter` network. This time, we mapped port `8080` to port `80` on the container. Note that we did not need to worry about exposing any ports of the Redis container. That is because the Redis image comes with some defaults that expose the default port, which is `6379` for us. This can be seen by running `docker container ls`:



```
~ docker container ls
CONTAINER ID        IMAGE                        COMMAND                  CREATED         STATU
S              PORTS                  NAMES
e7335ca1830d        russmckendrick/moby-counter   "node index.js"          3 minutes ago   Up 3
minutes        0.0.0.0:8080->80/tcp   moby-counter
f3b6a0d45f56        redis:alpine                  "docker-entrypoint.s…"   3 minutes ago   Up 3
minutes        6379/tcp               redis
~
```

Figure 4.28 – Listing the containers needed for our application

All that remains now is to access the application. To do this, open your browser and go to `http://localhost:8080/`. You should be greeted by a mostly blank page, with the message **Click to add logos…**:



Click to add logos...

Figure 4.29 – Our application is ready to go

Clicking anywhere on the page will add Docker logos, so click away:

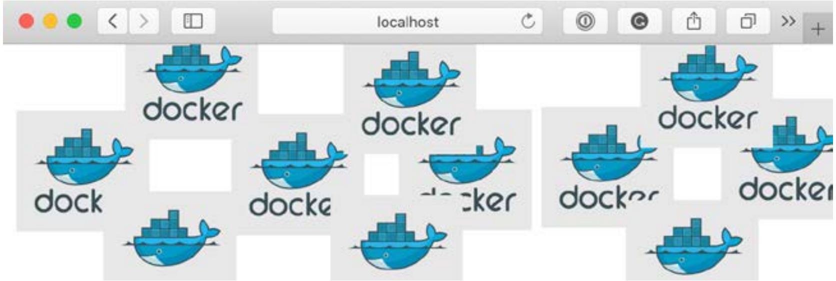


Figure 4.30 – Adding some logos to the page

So, what is happening? The application that is being served from the moby-counter container is making a connection to the redis container, and using the service to store the onscreen coordinates of each of the logos that you place on the screen by clicking.

How is the moby-counter application connecting to the redis container? Well, in the server.js file, the following default values are being set:

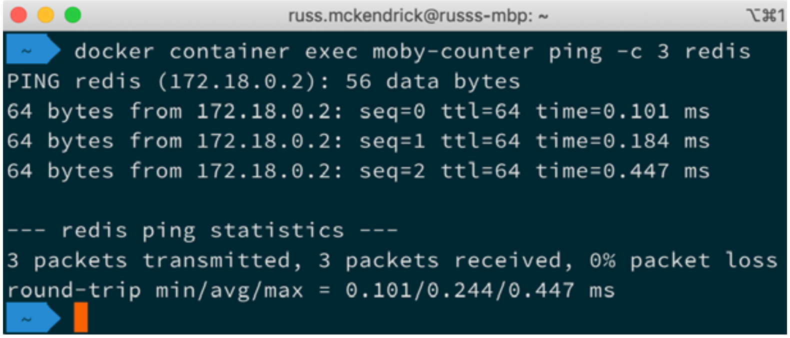
```
var port = opts.redis_port || process.env.USE_REDIS_PORT ||
6379
```

```
var host = opts.redis_host || process.env.USE_REDIS_HOST ||
'redis'
```

This means that the moby-counter application is looking to connect to a host called redis on port 6379. Let's try using the exec command to ping the redis container from the moby-counter application and see what we get:

```
$ docker container exec moby-counter ping -c 3 redis
```

You should see something similar to the following output:



```
~ docker container exec moby-counter ping -c 3 redis
PING redis (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.101 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.184 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.447 ms

--- redis ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.101/0.244/0.447 ms
~
```

Figure 4.31 – Pinging the redis container using the container name

As you can see, the moby-counter container resolves redis to the IP address of the redis container, which is 172.18.0.2. You may be thinking that the application's host file contains an entry for the redis container; let's take a look using the following command:

```
$ docker container exec moby-counter cat /etc/hosts
```

This returns the content of /etc/hosts, which, in my case, looks like the following:

```
    127.0.0.1    localhost
    ::1    localhost ip6-localhost ip6-loopback
    fe00::0     ip6-localnet
    ff00::0     ip6-mcastprefix
    ff02::1     ip6-allnodes
    ff02::2     ip6-allrouters
    172.18.0.3   e7335ca1830d
```

Other than the entry at the end, which is actually the IP address resolving to the hostname of the local container, e7335ca1830d is the ID of the container; there is no sign of an entry for redis. Next, let's check /etc/resolv.conf by running the following command:

```
$ docker container exec moby-counter cat /etc/resolv.conf
```

This returns what we are looking for. As you can see, we are using a local nameserver:

```
nameserver 127.0.0.11
options ndots:0
```

Let's perform a DNS lookup on redis against 127.0.0.11 using the following command:

```
$ docker container exec moby-counter nslookup redis 127.0.0.11
```

This returns the IP address of the redis container:

```
    Server:    127.0.0.11
    Address 1: 127.0.0.11

    Name:      redis
    Address 1: 172.18.0.2 redis.moby-counter
```

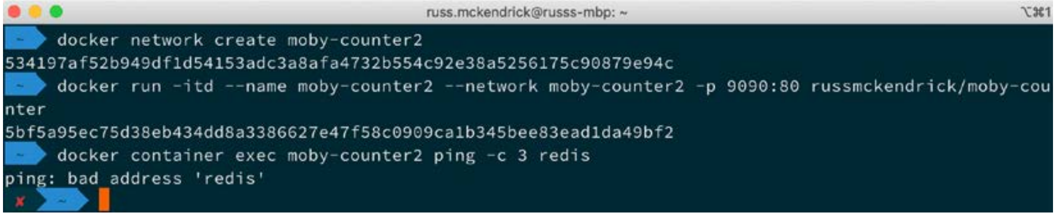
Let's create a second network and launch another application container:

```
$ docker network create moby-counter2
$ docker run -itd --name moby-counter2 --network moby-counter2
-p 9090:80 russmckendrick/moby-counter
```

Now that we have the second application container up and running, let's try pinging the `redis` container from it:

```
$ docker container exec moby-counter2 ping -c 3 redis
```

In my case, I get the following error:



```
~ docker network create moby-counter2
534197af52b949df1d54153adc3a8afa4732b554c92e38a5256175c90879e94c
~ docker run -itd --name moby-counter2 --network moby-counter2 -p 9090:80 russmckendrick/moby-cou
nter
5bf5a95ec75d38eb434dd8a3386627e47f58c0909ca1b345bee83ead1da49bf2
~ docker container exec moby-counter2 ping -c 3 redis
ping: bad address 'redis'
✘ ~
```

Figure 4.32 – Isolating our applications in different networks

Let's check the `resolv.conf` file to see whether the same `nameserver` is being used already, as follows:

```
$ docker container exec moby-counter2 cat /etc/resolv.conf
```

As you can see from the following output, the `nameserver` is indeed in use already:

```
    nameserver 127.0.0.11
    options ndots:0
```

As we have launched the `moby-counter2` container in a different network to that where the container named `redis` is running, we cannot resolve the hostname of the container:

```
$ docker container exec moby-counter2 nslookup redis 127.0.0.11
```

So, it returns a bad address error:

```
    Server:    127.0.0.11
    Address 1: 127.0.0.11

    nslookup: can't resolve 'redis': Name does not resolve
```

Let's look at launching a second Redis server in our second network. As we have already discussed, we cannot have two containers with the same name, so let's creatively name it `redis2`. As our application is configured to connect to a container that resolves to `redis`, does this mean we will have to make changes to our application container? No, Docker has you covered.

While you cannot have two containers with the same name, as we have already discovered, our second network is running completely isolated from our first network, meaning that we can still use the DNS name of `redis`. To do this, we need to add the `-network-alias` flag as follows:

```
$ docker container run -d --name redis2 --network moby-counter2
--network-alias redis redis:alpine
```

As you can see, we have named the container `redis2`, but set `--network-alias` to be `redis`:

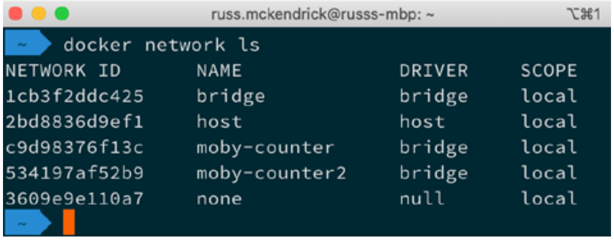
```
$ docker container exec moby-counter2 nslookup redis 127.0.0.1
```

This means that when we perform the lookup, we see the correct IP address returned:

```
Server:    127.0.0.1
Address 1: 127.0.0.1 localhost

Name:      redis
Address 1: 172.19.0.3 redis2.moby-counter2
```

As you can see, `redis` is actually an alias for `redis2.moby-counter2`, which then resolves to `172.19.0.3`.

Now we should have two applications running side by side in their own isolated networks on your local Docker host, accessible at `http://localhost:8080/` and `http://localhost:9090/`. Running `docker network ls` will display all of the networks configured on your Docker host, including the default networks:



```
~ docker network ls
NETWORK ID     NAME            DRIVER    SCOPE
1cb3f2ddc425   bridge          bridge    local
2bd8836d9ef1   host            host      local
c9d98376f13c   moby-counter    bridge    local
534197af52b9   moby-counter2   bridge    local
3609e9e110a7   none            null      local
~
```

Figure 4.33 – Listing our networks

You can find out more information about the configuration of the networks by running the following `inspect` command:

```
$ docker network inspect moby-counter
```

Running the preceding command returns the following JSON array. It starts by giving us some general information on the network:

```
[
    {
        "Name": "moby-counter",
        "Id": "c9d98376f13ccd556d84b708e132350900036fb4
cfecf275dcbd8657dc69b22c",
        "Created": "2020-03-29T13:06:03.3911316Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
```

Next up is the configuration used by the IP Address Management system. It shows the subnet range and gateway IP address:

```
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
```

What follows next is the remainder of the general configuration:

```
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
```

Then, we have details pertaining to the containers, which are attached to the network. This is where we can find the IP address and MAC address of each container:

```
        "Containers": {
            "e7335ca1830da66d4bdc2915a6a35e83e
546cbde63cd97ab48bfd3ca06ae99ae": {
                "Name": "moby-counter",
                "EndpointID": "fb405fac3e0814e3ab7f1b8e2c4
2bbfe09d751982c502ff196ac794e382bbb2a",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            },
            "f3b6a0d45f56fe2a0b54beb4b89d6094aaf
42598e11c3080ef0a21b78f0ec159": {
                "Name": "redis",
                "EndpointID": "817833e6bba40c73a3a349fae
53205b1c9e19d73f3a8d5e296729ed5876cf648",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
```

Finally, we have the last bit of the configuration:

```
        "Options": {},
        "Labels": {}
    }
]
```

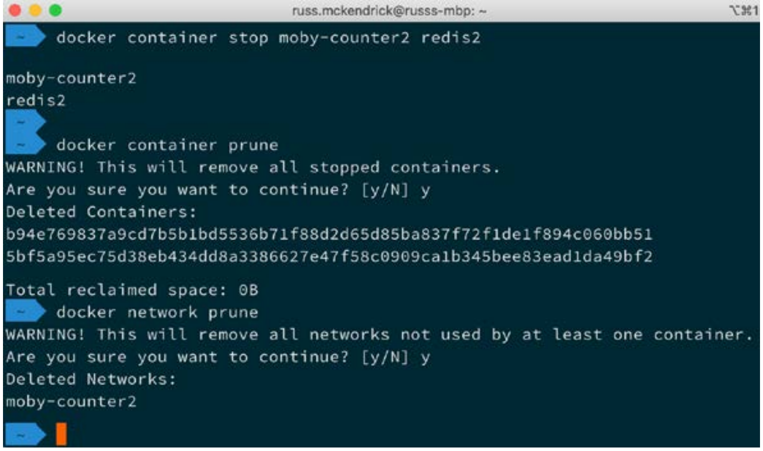
As you can see, it contains information on the network address being used in the IPAM section, along with details on each of the two containers running in the network.

> **Important note**
>
> **IP address management** (**IPAM**) is a means of planning, tracking, and managing IP addresses within the network. `IPAM` has both `DNS` and `DHCP` services, so each service is notified of changes in the other. For example, `DHCP` assigns an address to `container2`. The `DNS` service is then updated to return the IP address assigned by `DHCP` whenever a lookup is made against `container2`.

Before we progress to the next section, we should remove one of the applications and associated networks. To do this, run the following commands:

```
$ docker container stop moby-counter2 redis2
$ docker container prune
$ docker network prune
```

These will remove the containers and network, as shown in the following screenshot:



```
~  docker container stop moby-counter2 redis2

moby-counter2
redis2
~
~  docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
b94e769837a9cd7b5b1bd5536b71f88d2d65d85ba837f72f1de1f894c060bb51
5bf5a95ec75d38eb434dd8a3386627e47f58c0909ca1b345bee83ead1da49bf2

Total reclaimed space: 0B
~  docker network prune
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Networks:
moby-counter2

~
```

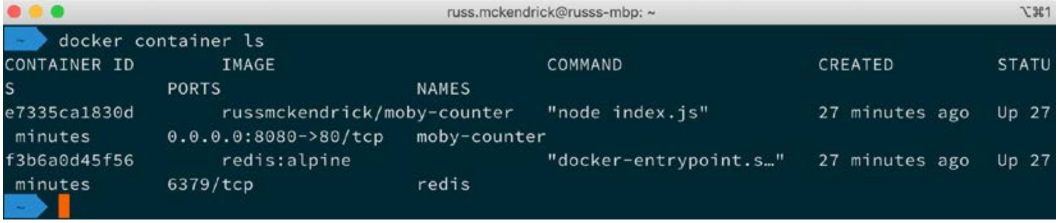
Figure 4.34 – Removing unused networks with the prune command

As mentioned at the start of this section, this is only the default network driver, meaning that we are restricted to our networks being available only on a single Docker host. In later chapters, we will look at how we can expand our Docker network across multiple hosts and even providers.

Now that we know the basics around Docker networking, let's take a look at how we can work with additional storage for our containers.

Docker volumes

If you have been following along with the network example from the previous section, you should have two containers running, as shown in the following screenshot:



```
~ docker container ls
CONTAINER ID        IMAGE                           COMMAND                  CREATED             STATU
S                   PORTS                  NAMES
e7335ca1830d        russmckendrick/moby-counter     "node index.js"          27 minutes ago      Up 27
 minutes            0.0.0.0:8080->80/tcp   moby-counter
f3b6a0d45f56        redis:alpine                    "docker-entrypoint.s…"   27 minutes ago      Up 27
 minutes            6379/tcp               redis
~
```

Figure 4.35 – Listing the running containers

When you go to the application in a browser (at `http://localhost:8080/`), you will probably see that there already are Docker logos on screen. Let's stop and then remove the Redis container and see what happens. To do this, run the following commands:

```
$ docker container stop redis
$ docker container rm redis
```

If you have your browser open, you may notice that the Docker icons have faded into the background and there is an animated loader in the center of the screen. This is basically to show that the application is waiting for the connection to the Redis container to be re-established:

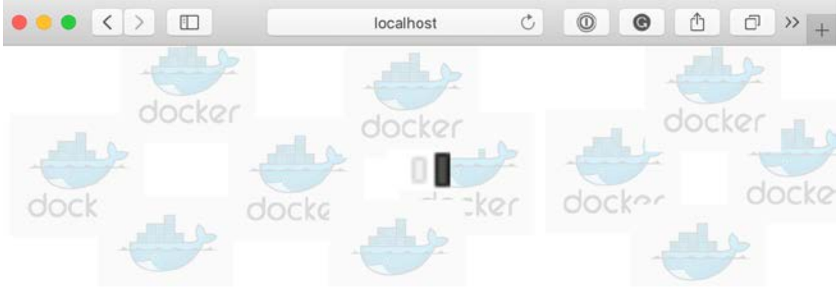


Figure 4.36 – The application can no longer connect to Redis

Relaunch the Redis container using the following command:

```
$ docker container run -d --name redis --network moby-counter
redis:alpine
```

This restores connectivity. However, when you start to interact with the application, your previous icons disappear and you are left with a clean slate. Quickly add some more logos to the screen, this time placed in a different pattern, as I have done here:

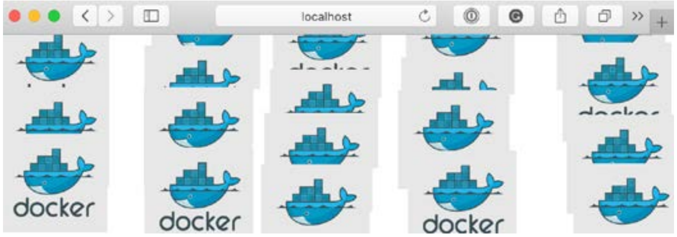


Figure 4.37 – Adding more logos

Once you have a pattern, let's remove the Redis container again by running the following commands:

```
$ docker container stop redis
$ docker container rm redis
```

As we discussed earlier in the chapter, losing the data in the container is to be expected. However, as we used the official Redis image, we haven't, in fact, lost any of our data.

The Dockerfile for the official Redis image that we used looks like the following:

```
FROM alpine:3.11

RUN addgroup -S -g 1000 redis && adduser -S -G redis -u 999
redis
RUN apk add --no-cache \
     'su-exec>=0.2' \
     tzdata

ENV REDIS_VERSION 5.0.8
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-
5.0.8.tar.gz
ENV REDIS_DOWNLOAD_SHA f3c7eac42f433326a8d981b50dba0169
fdfaf46abb23fcda2f933a7552ee4ed7
```

The preceding steps prepare the container by adding a group and user, installing a few packages, and setting some environment variables. The following steps install the prerequisites needed to run Redis:

```
RUN set -eux; \
    \
    apk add --no-cache --virtual .build-deps \
      coreutils \
      gcc \
      linux-headers \
      make \
      musl-dev \
      openssl-dev \
    ; \
    \
```

Now, the Redis source code is downloaded and copied to the right place on the image:

```
    wget -O redis.tar.gz "$REDIS_DOWNLOAD_URL"; \
    echo "$REDIS_DOWNLOAD_SHA *redis.tar.gz" | sha256sum -c -; \
    mkdir -p /usr/src/redis; \
    tar -xzf redis.tar.gz -C /usr/src/redis --strip-
components=1; \
    rm redis.tar.gz; \
    \
```

Now that the source for Redis is in the image, the configuration is applied:

```
    grep -q '^#define CONFIG_DEFAULT_PROTECTED_MODE 1$' /usr/
src/redis/src/server.h; \
    sed -ri 's!^(#define CONFIG_DEFAULT_PROTECTED_MODE) 1$!\1
0!' /usr/src/redis/src/server.h; \
    grep -q '^#define CONFIG_DEFAULT_PROTECTED_MODE 0$' /usr/
src/redis/src/server.h; \
    \
```

Now, Redis is compiled and tested:

```
    make -C /usr/src/redis -j "$(nproc)" all; \
    make -C /usr/src/redis install; \
    \
    serverMd5="$(md5sum /usr/local/bin/redis-server | cut -d' ' -f1)"; export serverMd5; \
    find /usr/local/bin/redis* -maxdepth 0 \
     -type f -not -name redis-server \
     -exec sh -eux -c ' \
         md5="$(md5sum "$1" | cut -d" " -f1)"; \
         test "$md5" = "$serverMd5"; \
     ' -- '{}' ';' \
     -exec ln -svfT 'redis-server' '{}' ';' \
    ; \
    \
```

The build directory is then removed and the packages that are no longer needed are removed:

```
  rm -r /usr/src/redis; \
  \
  runDeps="$( \
   scanelf --needed --nobanner --format '%n#p' --recursive /usr/local \
        | tr ',' '\n' \
        | sort -u \
        | awk 'system("[ -e /usr/local/lib/" $1 " ]") == 0 { next } { print "so:" $1 }' \
  )"; \
  apk add --no-network --virtual .redis-rundeps $runDeps; \
  apk del --no-network .build-deps; \
  \
```

Now that Redis is built and the packages and build artifacts tided up, a final test is run. If it fails here, the build will also fail:

```
  redis-cli --version; \
  redis-server --version
```

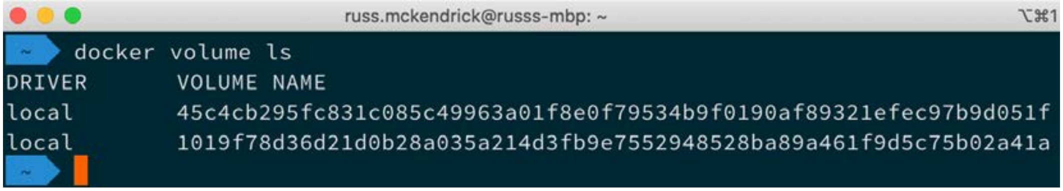
With everything installed, the final set of image configuration can take place:

```
RUN mkdir /data && chown redis:redis /data
VOLUME /data
WORKDIR /data

COPY docker-entrypoint.sh /usr/local/bin/
ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 6379
CMD ["redis-server"]
```

If you notice, during the last part of the file, there are the `VOLUME` and `WORKDIR` directives declared; this means that when our container was launched, Docker actually created a volume and then run `redis-server` from within the volume. We can see this by running the following command:

```
$ docker volume ls
```

This should show at least two volumes, as seen in the following screenshot:



```
~ docker volume ls
DRIVER        VOLUME NAME
local         45c4cb295fc831c085c49963a01f8e0f79534b9f0190af89321efec97b9d051f
local         1019f78d36d21d0b28a035a214d3fb9e7552948528ba89a461f9d5c75b02a41a
~
```

Figure 4.38 – Listing the volumes

As you can see, the volume name is not very friendly at all. In fact, it is the unique ID of the volume. So how can we use the volume when we launch our Redis container? We know from the Dockerfile that the volume was mounted at `/data` within the container, so all we have to do is tell Docker which volume to use and where it should be mounted at runtime.

To do this, run the following command, making sure you replace the volume ID with that of your own:

```
$ docker container run -d --name redis -v 45c4cb295fc831c085c49
963a01f8e0f79534b9
f0190af89321efec97b9d051f:/data -network moby-counter
redis:alpine
```

If your application page looks like it is still trying to reconnect to the Redis container once you have launched your Redis container, then you may need to refresh your browser. Failing that, restarting the application container by running `docker container restart moby-counter` and then refreshing your browser again should work.

You can view the contents of the volume by running the following command to attach the container and list the files in `/data`:

```
$ docker container exec redis ls -lhat /data
```

This will return something that looks like the following:

```
    total 12K
    drwxr-xr-x    1 root     root         4.0K Mar 29 13:51 ..
    drwxr-xr-x    2 redis    redis        4.0K Mar 29 13:35 .
    -rw-r--r--    1 redis    redis         210 Mar 29 13:35
dump.rdb
```

You can also remove your running container and relaunch it, but this time using the ID of the second volume. As you can see from the application in your browser, the two different patterns you originally created are intact.

Let's remove the `Redis` container again:

```
$ docker container stop redis
$ docker container rm redis
```

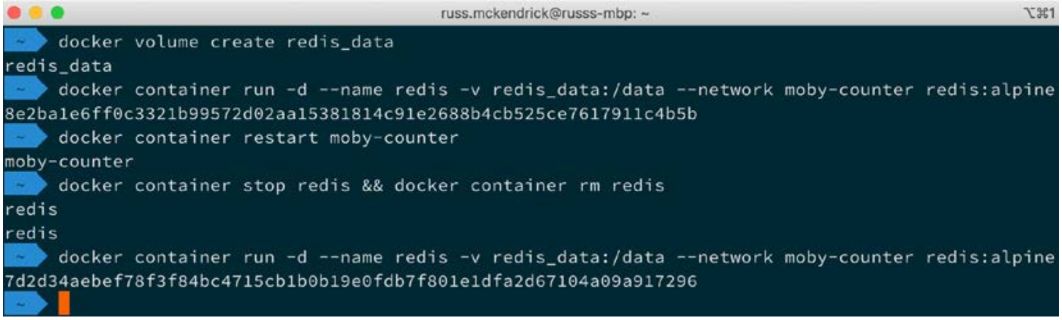
Finally, you can override the volume with your own. To create a volume, we need to use the `volume` command:

```
$ docker volume create redis_data
```

Once created, we will be able to use the `redis_data` volume to store our `Redis` by running the following command after removing the `redis` container, which is probably already running:

```
$ docker container run -d --name redis -v redis_data:/data
--network moby-counter redis:alpine
```

We can then reuse the volume as needed. The following screen shows the volume being created, attached to a container that is then removed and finally reattached to a new container:



```
~ docker volume create redis_data
redis_data
~ docker container run -d --name redis -v redis_data:/data --network moby-counter redis:alpine
8e2ba1e6ff0c3321b99572d02aa15381814c91e2688b4cb525ce7617911c4b5b
~ docker container restart moby-counter
moby-counter
~ docker container stop redis && docker container rm redis
redis
redis
~ docker container run -d --name redis -v redis_data:/data --network moby-counter redis:alpine
7d2d34aebef78f3f84bc4715cb1b0b19e0fdb7f801e1dfa2d67104a09a917296
~
```

Figure 4.39 – Creating a volume and attaching it to a container

Like the `network` command, we can view more information on the volume using the `inspect` command, as follows:

```
$ docker volume inspect redis_data
```

The preceding command will produce something like the following output:

```
[
    {
        "CreatedAt": "2020-03-29T14:01:05Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/redis_data/_
data",
        "Name": "redis_data",
        "Options": {},
        "Scope": "local"
    }
]
```

You can see that there is not much to a volume when using the local driver. One interesting thing to note is that the path to where the data is stored on the Docker host machine is `/var/lib/docker/volumes/redis_data/_data`. If you are using Docker for Mac or Docker for Windows, then this path will be your Docker host virtual machine, and not your local machine, meaning that you do not have direct access to the data inside the volume.

Don't worry though; we will be looking at Docker volumes and how you can interact with data in later chapters. Before we tidy up our containers, networks, and volume, if you are running Docker Desktop, then we should take a look at the Docker Desktop Dashboard.

Docker Desktop Dashboard

If you are running either Docker for Mac or Docker for Windows, then there is an option within the main menu to open a dashboard that will display information on your running containers:

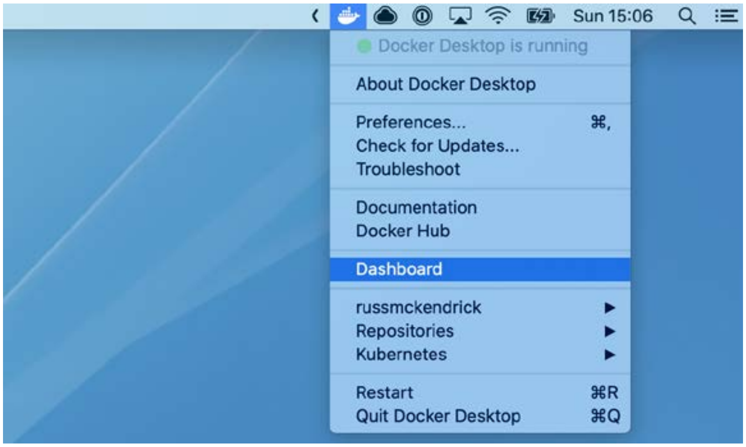
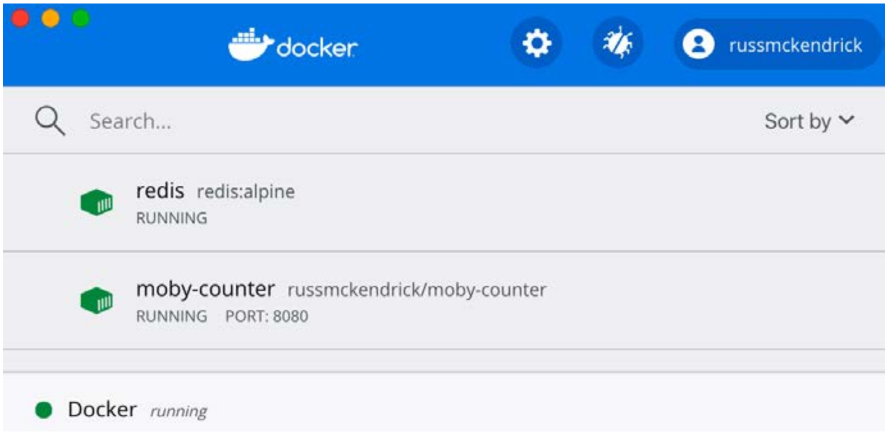


Figure 4.40 – Opening the Docker Desktop Dashboard

Once open, you should see something like the following screen. As you can see, we have our `redis` and `moby-counter` containers listed:



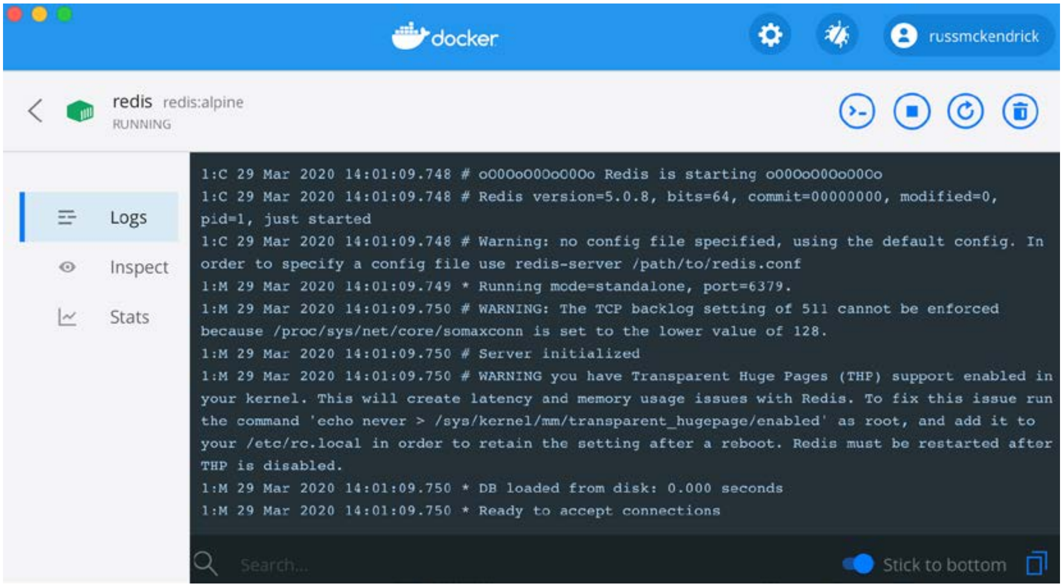
redis redis:alpine
RUNNING

moby-counter russmckendrick/moby-counter
RUNNING PORT: 8080

● Docker *running*

Figure 4.41 – Viewing the running containers

Selecting the `redis` container will take you to an overview screen that defaults to the `Logs` output:



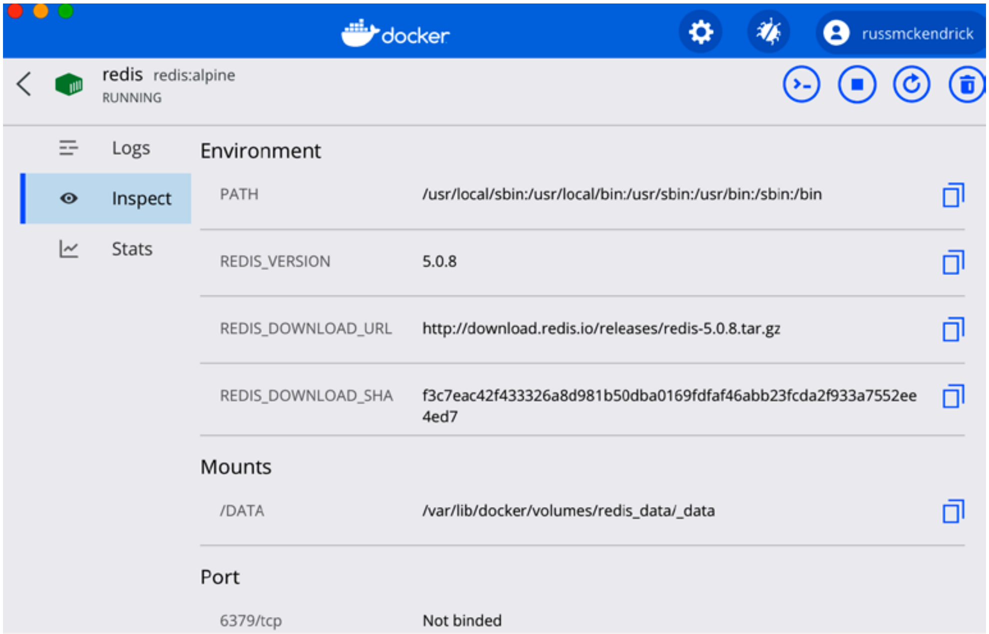
```
1:C 29 Mar 2020 14:01:09.748 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 29 Mar 2020 14:01:09.748 # Redis version=5.0.8, bits=64, commit=00000000, modified=0,
pid=1, just started
1:C 29 Mar 2020 14:01:09.748 # Warning: no config file specified, using the default config. In
order to specify a config file use redis-server /path/to/redis.conf
1:M 29 Mar 2020 14:01:09.749 * Running mode=standalone, port=6379.
1:M 29 Mar 2020 14:01:09.750 # WARNING: The TCP backlog setting of 511 cannot be enforced
because /proc/sys/net/core/somaxconn is set to the lower value of 128.
1:M 29 Mar 2020 14:01:09.750 # Server initialized
1:M 29 Mar 2020 14:01:09.750 # WARNING you have Transparent Huge Pages (THP) support enabled in
your kernel. This will create latency and memory usage issues with Redis. To fix this issue run
the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to
your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after
THP is disabled.
1:M 29 Mar 2020 14:01:09.750 * DB loaded from disk: 0.000 seconds
1:M 29 Mar 2020 14:01:09.750 * Ready to accept connections
```

Figure 4.42 – Overview screen of the Logs output

Let's start at the top of the screen. To the right here, you can see four blue icons; these are as follows, from left to right:

- **Connect to container**: This will open your default Terminal application and connect to the currently selected container.
- **Stop the currently connected container**: When stopped, the icon will change to a **Start** icon.
- Next, we have the **Restart icon**. Clicking this will, well you guessed it right?! It will restart the currently selected container.
- The final **Trash icon** will terminate the currently selected container.

Next, we have the menu items on the left-hand side of the screen. We have already seen the **Logs** output; this is updated in real time and you also have the option of searching through the log output. Below that we have **Inspect**; this displays some basic information about the container:



redis redis:alpine
RUNNING

Environment

PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
REDIS_VERSION	5.0.8
REDIS_DOWNLOAD_URL	http://download.redis.io/releases/redis-5.0.8.tar.gz
REDIS_DOWNLOAD_SHA	f3c7eac42f433326a8d981b50dba0169fdfaf46abb23fcda2f933a7552ee4ed7

Mounts

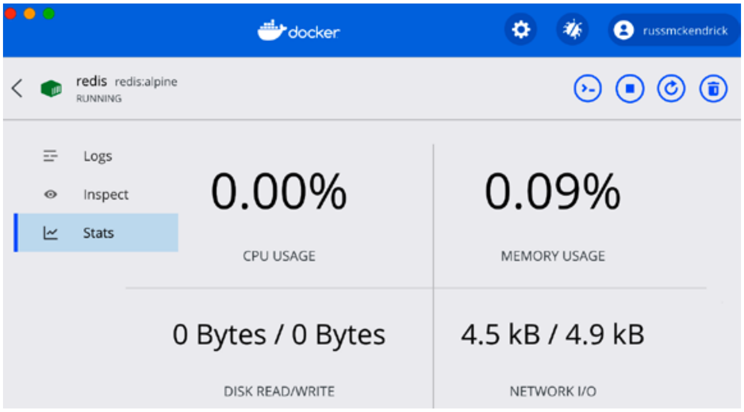
/DATA	/var/lib/docker/volumes/redis_data/_data

Port

6379/tcp	Not binded

Figure 4.43 – Getting information on the container using inspect

The final item is `Stats`; this – as you may have already figured out, gives us the same output as the `docker container stats redis` command:



redis redis:alpine
RUNNING

0.00%
CPU USAGE

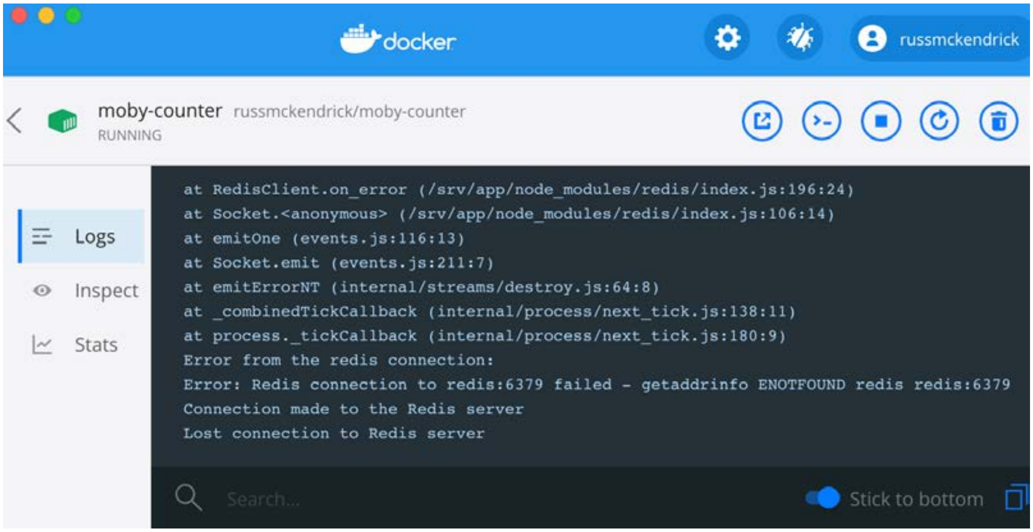
0.09%
MEMORY USAGE

0 Bytes / 0 Bytes
DISK READ/WRITE

4.5 kB / 4.9 kB
NETWORK I/O

Figure 4.44 – Viewing the real-time stats

Going to the `moby-counter` container adds an additional icon to the start of the top menu:



```
at RedisClient.on_error (/srv/app/node_modules/redis/index.js:196:24)
at Socket.<anonymous> (/srv/app/node_modules/redis/index.js:106:14)
at emitOne (events.js:116:13)
at Socket.emit (events.js:211:7)
at emitErrorNT (internal/streams/destroy.js:64:8)
at _combinedTickCallback (internal/process/next_tick.js:138:11)
at process._tickCallback (internal/process/next_tick.js:180:9)
Error from the redis connection:
Error: Redis connection to redis:6379 failed - getaddrinfo ENOTFOUND redis redis:6379
Connection made to the Redis server
Lost connection to Redis server
```

Figure 4.45 – Viewing the additional icon

This will open your default browser and take you to the externally exposed port, which in this case is `http://localhost:8080`.

You have noticed that there are some features, such as the ability to create containers, in the dashboard. However, as new versions are released, I am sure that more management features will be added.

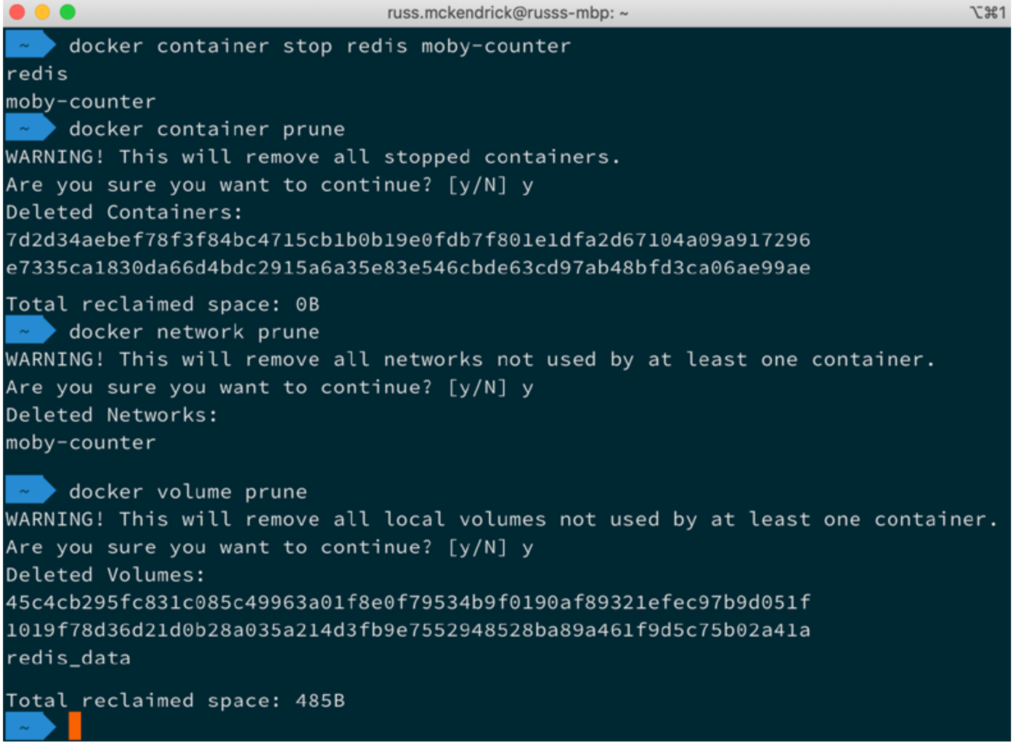
Now, we should tidy up. First of all, remove the two containers and network:

```
$ docker container stop redis moby-counter
$ docker container prune
$ docker network prune
```

Then, we can remove the volumes by running the following command:

```
$ docker volume prune
```

You should see something similar to the following Terminal output:



```
~ docker container stop redis moby-counter
redis
moby-counter
~ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
7d2d34aebef78f3f84bc4715cb1b0b19e0fdb7f801e1dfa2d67104a09a917296
e7335ca1830da66d4bdc2915a6a35e83e546cbde63cd97ab48bfd3ca06ae99ae

Total reclaimed space: 0B
~ docker network prune
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Networks:
moby-counter

~ docker volume prune
WARNING! This will remove all local volumes not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Volumes:
45c4cb295fc831c085c49963a01f8e0f79534b9f0190af89321efec97b9d051f
1019f78d36d21d0b28a035a214d3fb9e7552948528ba89a461f9d5c75b02a41a
redis_data

Total reclaimed space: 485B
~
```

Figure 4.46 – Removing everything we have launched

We are now back to having a clean slate, so we can progress to the next chapter.

Summary

In this chapter, we looked at how you can use the Docker command-line client to both manage individual containers and launch multi-container applications in their own isolated Docker networks. We also discussed how we can persist data on the filesystem using Docker volumes. So far, in this and previous chapters, we have covered in detail the majority of the available commands that we will use in the following sections:

```
$ docker container [command]
$ docker network [command]
$ docker volume [command]
$ docker image [command]
```

Now that we have covered the four main areas of using Docker locally, we can start to look at how to create more complex applications. In the next chapter, we will take a look at another core Docker tool, called **Docker Compose**.

Questions

1. Which flag do you have to append to `docker container ls` to view all the containers, both running and stopped?
2. True or false: the `-p 8080:80` flag will map port `80` on the container to port `8080` on the host.
3. Explain the difference between what happens when you use *Ctrl* + *C* to exit a container you have attached, compared to using the `attach` command with `--sig-proxy=false`.
4. True or false: The `exec` command attaches you to the running process.
5. Which flag would you use to add an alias to a container so that it responds to DNS requests, when you already have a container running with the same DNS name in another network?
6. Which command would you use to find out details on a Docker volume?

Further reading

You can find out more about some of the topics we have discussed in this chapter at the following links:

- The names generator code: `https://github.com/moby/moby/blob/master/pkg/namesgenerator/names-generator.go`
- The `cgroups` freezer function: `https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt`
- Redis: `https://redis.io/`

5
Docker Compose

In this chapter, we will be taking a look at another core Docker tool called Docker Compose, and also the currently in-development Docker App.

Both of these tools allow you to easily define, build, and distribute applications that are made up of more than one container, using syntax similar to the commands we have been using to manually launch our containers in previous chapters.

We are going to break the chapter down into the following sections:

- Exploring the basics of Docker Compose
- Making our first Docker Compose application
- Exploring Docker Compose commands
- Using Docker App

Technical requirements

As in previous chapters, we will continue to use our local Docker installations, and the screenshots in this chapter will again be from my preferred operating system, macOS. As before, the Docker commands we will be running will work on all three of the operating systems on which we have installed Docker so far. However, some of the supporting commands, which will be few and far between, may only apply to macOS- and Linux-based operating systems—these will be highlighted.

Exploring the basics of Docker Compose

In *Chapter 1, Docker Overview*, we discussed a few of the problems that Docker has been designed to solve. We explored how Docker addresses the challenges faced by a lot of development and operations teams. One such solution was to run two different application stacks side by side by isolating each application stack's processes into a single container. This lets you run two entirely different versions of the same software stack—let's say PHP 5.6 and PHP 7—on the same host, as we did in *Chapter 2, Building Container Images*.

Toward the end of *Chapter 4, Managing Containers*, we launched an application that was made up of multiple containers rather than running the required software stack in a single container. The example application we started, `Moby Counter`, is written in Node.js and uses Redis as a backend to store key values, which in our case were the coordinates of the Docker logos on screen.

To be able to run the Moby Counter application, we had to launch two different containers, one for the Node.js application and one for Redis. While it was quite simple to do this as the application itself was quite basic, there are a number of disadvantages to manually launching single containers.

For example, if I wanted a colleague to deploy the same application, I would have to pass them the following commands:

```
$ docker image pull redis:alpine
$ docker image pull russmckendrick/moby-counter
$ docker network create moby-counter
$ docker container run -d --name redis --network moby-counter redis:alpine
$ docker container run -d --name moby-counter --network moby-counter -p 8080:80 russmckendrick/moby-counter
```

Admittedly, I could get away with losing the first two commands as the image would be pulled down during the two `docker run` commands if the images were not present on my colleague's local machine, but as the applications start to get more complex, I will have to start passing on an ever-growing set of commands and instructions.

I would also have to make it clear that they would have to take into account the order in which the commands need to be executed. Furthermore, my notes would have to include details of any potential issues to support them through any problems—which could mean we find ourselves in a "worked fine in Dev, Ops problem now" scenario, which we want to avoid at all costs.

While Docker's responsibility should end at creating the images and launching containers using these images, Docker's creators anticipated such a scenario and sought to overcome this. Thanks to Docker, people no longer have to worry about inconsistencies in the environment in which they are launching their applications, as these can now be shipped in images.

Orchard Laboratories

Before we look at Docker Compose, let's take a quick step back in time to July 2014, when Docker purchased a small British start-up called Orchard Laboratories who offered two container-based products.

The first of the two products was a Docker-based hosting platform. From a single command, `orchard`, you could launch a host machine and then proxy your Docker commands through to the newly launched host; for example, you would use the following commands:

```
$ orchard hosts create
$ orchard docker run -p 6379:6379 -d orchardup/redis
```

These commands would have launched a Docker host on Orchard's platform, and then a Redis container. I say *would have*, as one of the first things Docker did when they purchased Orchard Laboratories was to retire the Orchard hosting service.

The second Orchard Laboratories offering was an open source command-line tool called Fig, and it is this is what Docker had their eyes on when they purchased Orchard Laboratories. Fig, which was written in Python, let you use a YAML file to define how you would like your multi-container application to be structured. Fig took the YAML file and instructed Docker to launch the containers as defined.

The advantage of this was that because it was a YAML file, it was straightforward for developers to start shipping `fig.yml` files alongside their Dockerfiles within their code bases. A typical `fig.yml` file would have looked like the following:

```
web:
  image: web
  links:
   - db
  ports:
   - "8000:8000"
   - "49100:22"
db:
  image: postgres
```

To launch the two containers defined in the `fig.yml` file, you would have had to have run the following command from inside the same folder where the `fig.yml` file was stored:

```
$ fig up
```

You may have noticed that I have been referring to Fig in the past tense, and that is because in February 2015, Fig became Docker Compose. In the next section of this chapter, we are going to be looking at launching our first Docker Compose application, and one of the first things you will notice is how close the syntax for defining the application is to the original Fig syntax.

Making our first Docker Compose application

As part of our installation of Docker for Mac, Docker for Windows, and Docker on Linux in *Chapter 1, Docker Overview*, we installed Docker Compose, so rather than discussing what it does any further, let's try to bring up the two-container application we launched manually at the end of the last chapter, using just Docker Compose.

As already mentioned, Docker Compose uses a YAML file, typically named `docker-compose.yml`, to define what your multi-container application should look like. The Docker Compose representation of the two-container application we launched in *Chapter 4, Managing Containers* is as follows:

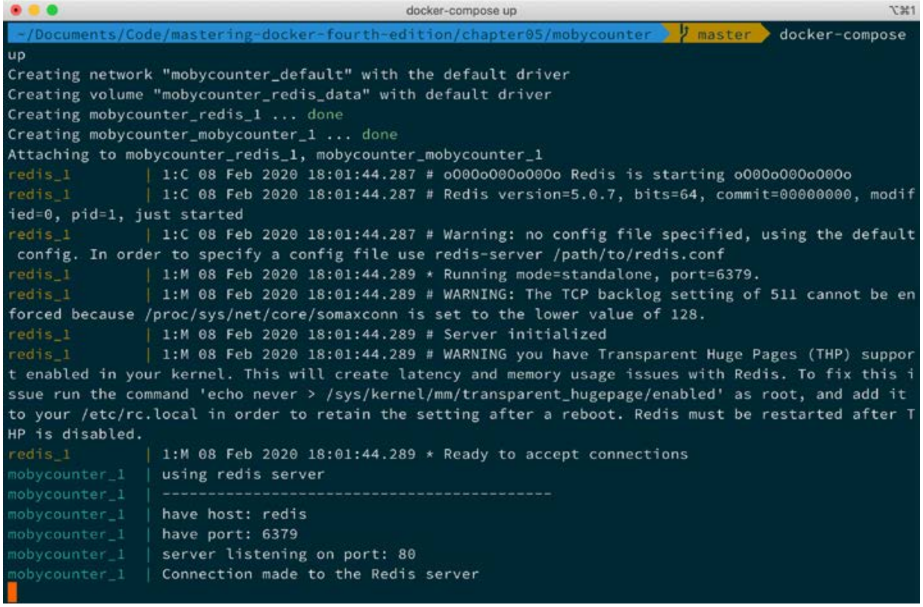
```
version: "3.7"

services:
    redis:
        image: redis:alpine
        volumes:
            - redis_data:/data
        restart: always
    mobycounter:
        depends_on:
            - redis
        image: russmckendrick/moby-counter
        ports:
            - "8080:80"
        restart: always

volumes:
    redis_data:
```

Even without working through each of the lines in the file, it should be quite straightforward to follow along with what is going on based on the commands we have been using throughout the previous chapters—we will be looking at the contents of the `docker-compose.yml` file in the next section of this chapter.

To launch our application, we simply change to the folder that contains your `docker-compose.yml` file and run the following:

```
$ docker-compose up
```

As you can see from the following Terminal output, quite a bit happened as Docker Compose launched our application:



```
~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master  docker-compose
up
Creating network "mobycounter_default" with the default driver
Creating volume "mobycounter_redis_data" with default driver
Creating mobycounter_redis_1 ... done
Creating mobycounter_mobycounter_1 ... done
Attaching to mobycounter_redis_1, mobycounter_mobycounter_1
redis_1          | 1:C 08 Feb 2020 18:01:44.287 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis_1          | 1:C 08 Feb 2020 18:01:44.287 # Redis version=5.0.7, bits=64, commit=00000000, modif
ied=0, pid=1, just started
redis_1          | 1:C 08 Feb 2020 18:01:44.287 # Warning: no config file specified, using the default
 config. In order to specify a config file use redis-server /path/to/redis.conf
redis_1          | 1:M 08 Feb 2020 18:01:44.289 * Running mode=standalone, port=6379.
redis_1          | 1:M 08 Feb 2020 18:01:44.289 # WARNING: The TCP backlog setting of 511 cannot be en
forced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1          | 1:M 08 Feb 2020 18:01:44.289 # Server initialized
redis_1          | 1:M 08 Feb 2020 18:01:44.289 # WARNING you have Transparent Huge Pages (THP) suppor
t enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this i
ssue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it
to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after T
HP is disabled.
redis_1          | 1:M 08 Feb 2020 18:01:44.289 * Ready to accept connections
mobycounter_1    | using redis server
mobycounter_1    | ------------------------------------------
mobycounter_1    | have host: redis
mobycounter_1    | have port: 6379
mobycounter_1    | server listening on port: 80
mobycounter_1    | Connection made to the Redis server
```

Figure 5.1 – Output of docker-compose up

You can see in the first few lines that Docker Compose did the following:

- It created a volume called `mobycounter_redis_data`, using the default driver we defined at the end of the `docker-compose.yml` file.
- It created a network called `mobycounter_default` using the default network driver—at no point did we ask Docker Compose to do this.
- It launched two containers, one called `mobycounter_redis_1` and another called `mobycounter_mobycounter_1`.

You may have also spotted that the Docker Compose namespace in our multi-container application has prefixed everything with `mobycounter`—it took this name from the folder our Docker Compose file was being stored in.

Once launched, Docker Compose attached to `mobycounter_redis_1` and `mobycounter_mobycounter_1` and streamed the output to our Terminal session. On the Terminal screen, you can see both `redis_1 and mobycounter_1` starting to interact with each other.

When running Docker Compose using `docker-compose up`, it will run in the foreground. Pressing *Ctrl* + *C* will stop the containers and return access to your Terminal session.

Docker Compose YAML file

Before we look at using Docker Compose more, we should have a deeper dive into `docker-compose.yml` files as these are at the heart of Docker Compose.

> **Important note**
>
> **YAML** is a recursive acronym that stands for **YAML Ain't Markup Language**. It is used by many different applications for both configuration and for defining data in a human-readable structured data format. The indentation you see in the examples is very important as it helps to define the structure of the data.

The Moby counter application

The `docker-compose.yml` file we used to launch our multi-container application is split into three separate sections.

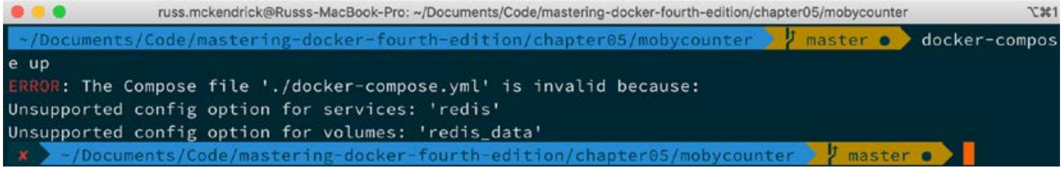
The first section simply specifies which version of the Docker Compose definition language we are using; in our case, as we are running a recent version of Docker and Docker Compose, we are using version 3, as illustrated here:

```
version: "3.7"
```

As already mentioned, Docker Compose has been around since 2015, during which time there have been quite a few different versions of both the Docker client and engine. As the software has been updated to include new features, the existing functionality has been streamlined to make it more performant, and also, some functionality has been split out of the core Docker engine or removed altogether. Docker Compose has been maintained to remain compatible with previsions versions of Docker.

If the version number is not declared at the start of the `docker-compose.yml` file, then Docker Compose will default to version 1; this is quite close to the original Fig syntax, meaning that Docker Compose will not be able to read our `docker-compose.yml` file as containers were defined under the `services` section and there is no support for volumes, networks, or even build arguments—all of which we will cover later in the chapter.

The following screenshot shows an example of what happens if we were to remove the version number:



russ.mckendrick@Russs-MacBook-Pro: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter
⌥⌘1

```
~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master ●  docker-compos
e up
ERROR: The Compose file './docker-compose.yml' is invalid because:
Unsupported config option for services: 'redis'
Unsupported config option for volumes: 'redis_data'
 ✘  ~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master ● 
```

Figure 5.2 – Output of docker-compose up

As you can see, trying to run `docker-compose up` without the version declared is going to end with errors, as Docker Compose quite literally doesn't know how to interpret the content we have defined.

The next section is where our containers are defined; this section is the `services` section. It takes the following format:

```
services:
----> container name:
--------> container options
------------> sub options
----> container name:
--------> container options
------------> sub options
```

As you can see, the `services` declaration has no indentation at all, and then each container has 4 spaces, with each of the options having 8 spaces; further options would then have 12 spaces. The number of spaces is a personal choice as I find it helps make it more readable—the important thing is to use spaces and not tabs, and make sure that your indentation is consistent throughout the file as this is used to clearly define blocks of information.

In our example, we defined two containers under the `services` section. In the following code snippet, they have been separated out to make it easy to read:

```
services:

    redis:
        image: redis:alpine
        volumes:
           - redis_data:/data
        restart: always

    mobycounter:
        depends_on:
            - redis
        image: russmckendrick/moby-counter
        ports:
            - "8080:80"
        restart: always
```

At first glance, the syntax for defining the service appears close to how you would launch a container using the `docker container run` command. I say "close" because although it makes perfect sense when you read the definition, it is only on closer inspection that you realize there is actually a lot of difference between the Docker Compose syntax and the `docker container run` command.

For example, there are no flags for the following when running the `docker container run` command:

- `image`: This tells Docker Compose which image to download and use. This does not exist as an option when running `docker container run` on the command line
  as you can only run a single container; as we have seen in previous chapters, the image is always defined toward the end of the command, without the need for a flag to be passed.
- `volume`: This is the equivalent of the `--volume` flag, but it can accept multiple volumes. It only uses the volumes that are declared in the Docker Compose YAML file; more on that in a moment.
- `depends_on`: This would never work as a `docker container run` invocation because the command is only targeting a single container. When it comes to Docker Compose, `depends_on` is used to help build some logic into the order your containers are launched in—for example, only launch container B when container A has successfully started.
- `ports`: This is basically the `--publish` flag, which accepts a list of ports.

The only part of the command we used that has an equivalent flag when running `docker container run` is `restart`. This is the same as using the `--restart` flag and accepts the same input.

The final section of our Docker Compose YAML file is where we declare our volumes, as follows:

```
volumes:
    redis_data:
```

This is the equivalent of running `docker volume create redis_data` using the Docker command-line client.

Example voting application

As mentioned already, the Docker Compose file for the Moby counter application is quite a simple example. Let's take a look at a more complex Docker Compose file and see how we can introduce building containers and multiple networks.

> **Important note**
>
> In the repository for this book, you will find a folder in the `chapter05` directory called `example-voting-app`. This is a fork of the voting application from the official Docker sample repository.

As you can see, if you were to open up the `docker-compose.yml` file, the application is made up of five containers, two networks, and a single volume. If we were to visualize the application, it would look something like the following:

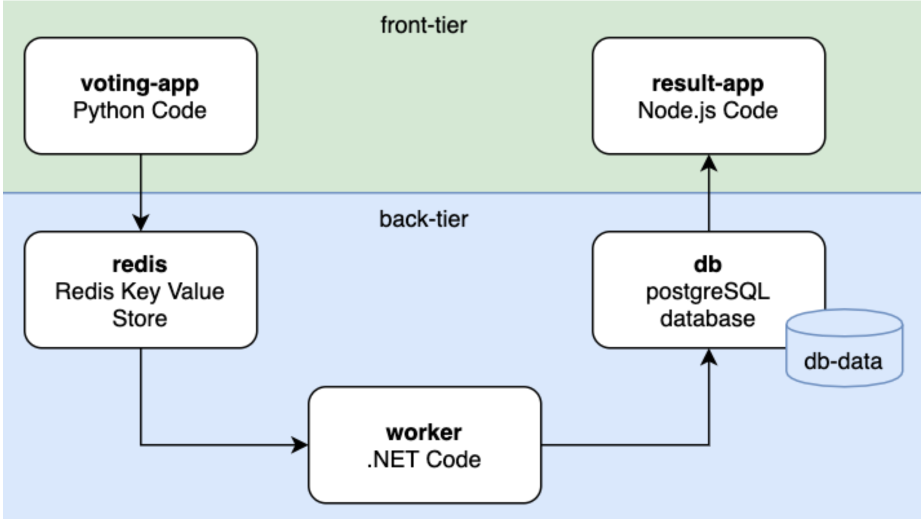


Figure 5.3 – Container structure for docker-compose.yml

Ignore the other files as we will look at some of these in future chapters; let's walk through the `docker-compose.yml` file as there is a lot going on. Have a look at the following code snippet:

```
version: "3"

services:
```

As you can see, it starts simply enough by defining the version, and then it starts to list the `service`. Our first container is called vote; it is a Python application that allows users to submit their vote. As you can see from the following definition, rather than downloading an image that contains the application, we are actually building an image by deploying our application using `build` instead of the `image` definition:

```
    vote:
        build: ./vote
        command: python app.py
        volumes:
            - ./vote:/app
        ports:
            - "5000:80"
        networks:
            - front-tier
            - back-tier
```

The `build` instruction here tells Docker Compose to build a container using the Dockerfile, which can be found in the `./vote` folder. The Dockerfile itself is quite straightforward for a Python application.

Once the container launches, we are then mounting the `./vote` folder from our host machine into the container, which is achieved by passing the path of the folder we want to mount and where within the container we would like it mounted.

We are telling the container to run the `python app.py` command when the container launches; we are mapping port `5000` on our host machine to port `80` on the container; and, finally, we are further attaching two networks to the container—one called `front-tier` and another called `back-tier`.

The `front-tier` network will have the containers that require ports to be mapped to the host machine; the `back-tier` network is reserved for containers that do not need their ports to be exposed and acts as a private, isolated network.

Next up, we have another container that is connected to the `front-tier` network. This container displays the results of the vote. The `result` container contains a Node.js application that connects to the PostgreSQL database (which we will get to in a moment) and displays the results in real time as votes are cast in the `vote` container. As with the `vote` container, the image is built locally using a Dockerfile that can be found in the `./result` folder, as illustrated in the following code snippet:

```
  result:
      build: ./result
      command: nodemon server.js
      volumes:
          - ./result:/app
      ports:
          - "5001:80"
          - "5858:5858"
       networks:
          - front-tier
          - back-tier
```

We are exposing port `5001`, which is where we can connect to see the results. The next—and final—application container is called `worker`, as illustrated in the following code snippet:

```
  worker:
      build:
          context: ./worker
      depends_on:
          - "redis"
          - "db"
      networks:
          - back-tier
```

The `worker` container runs a .NET application whose only job is to connect to Redis and register each vote by transferring it into a PostgreSQL database running on a container called `db`. The container is again built using a Dockerfile, but this time, rather than passing the path to the folder where the Dockerfile and application are stored, we are using `context`. This sets the working directory for the `docker build` command and also allows you to define additional options such as labels and changing the name of the Dockerfile.

As this container is doing nothing other than connecting to redis and the db container, it does not need any ports exposed as it has nothing connecting directly to it; it also does not need to communicate with either of the containers running on the front-tier network, meaning we just have to add the back-tier network.

So, we now have the vote application, which registers the votes from the end users and sends them to the redis container, where the vote is then processed by the worker container. The service definition for the redis container looks like the following:

```
    redis:
        image: redis:alpine
        container_name: redis
        ports: ["6379"]
        networks:
            - back-tier
```

This container uses the official Redis image and is not built from a Dockerfile; we are making sure that port 6379 is available, but only on the back-tier network. We are also specifying the name of the container, setting it to redis by using container_name. This is to avoid us having to make any considerations on the default names generated by Docker Compose within our code since, if you remember, Docker Compose uses the folder name to launch the containers in their own application namespace.

The next—and final—container is the PostgreSQL one (which we have already mentioned), called db, as illustrated in the following code snippet:

```
     db:
        image: postgres:9.4
        container_name: db
        volumes:
            - "db-data:/var/lib/postgresql/data"
        networks:
            - back-tier
```

As you can see, it looks quite similar to the redis container in that we are using the official image; however, you may notice that we are not exposing a port, as this is a default option in the official image. We are also specifying the name of the container.

As this is where our votes will be stored, we are creating and mounting a volume to act as persistent storage for our PostgreSQL database, as follows:

```
volumes:
    db-data:
```

Then, finally, here are the two networks we have been speaking about throughout when defining our application containers:

```
networks:
    front-tier:
    back-tier:
```

Running `docker-compose up` gives a lot of feedback on what is happening during the launch; it takes about 5 minutes to launch the application for the first time. If you are not following along and launching the application yourself, what follows is an abridged version of the launch.

> **TIP**
>
> You may get an error that states `npm ERR! request to https://registry.npmjs.org/nodemon failed, reason: Hostname/IP doesn't match certificate's altnames`. If you do, then run the `echo "104.16.16.35 registry.npmjs.org" >> /etc/hosts` command as a user with privileges to write to the `/etc/hosts` file on your machine.

Docker Compose starts by creating the networks and getting the volume ready for our containers to use, as illustrated here:

```
Creating network "example-voting-app_front-tier" with the default driver
Creating network "example-voting-app_back-tier" with the default driver
Creating volume "example-voting-app_db-data" with default driver
```

It then builds the `vote` container image, as follows:

```
Building vote
Step 1/7 : FROM python:2.7-alpine
2.7-alpine: Pulling from library/python
```

```
c9b1b535fdd9: Already exists
fea5c17ab132: Pull complete
5dbe995357bf: Pull complete
b6d238951af6: Pull complete
Digest: sha256:5217b150a5f7eecea55f6224440f3b5
c5f975edc32de7c0bfdf98280ed11d76c
Status: Downloaded newer image for python:2.7-alpine
```

Now that the images have been downloaded, the building of the first part of the application can start, as follows:

```
 ---> 7ec8514e7bc5
Step 2/7 : WORKDIR /app
 ---> Running in 7d26310faa98
Removing intermediate container 7d26310faa98
 ---> 8930ad501196
Step 3/7 : ADD requirements.txt /app/requirements.txt
 ---> 33ff980bd133
Step 4/7 : RUN pip install -r requirements.txt
 ---> Running in 999e575570ef
[lots of python build output here]
Removing intermediate container 999e575570ef
 ---> 72637119e7df
Step 5/7 : ADD . /app
 ---> 81adb9e92ce4
Step 6/7 : EXPOSE 80
 ---> Running in a5aaf5b9ed1b
Removing intermediate container a5aaf5b9ed1b
 ---> 366d2e32ceb4
Step 7/7 : CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80",
"--log-file", "-", "--access-logfile", "-", "--workers", "4",
"--keep-alive", "0"]
 ---> Running in 212e82c06cf3
Removing intermediate container 212e82c06cf3
 ---> 4553ffa35ea4
Successfully built 4553ffa35ea4
```

Now that the image has been built it will be tagged, as illustrated here:

```
Successfully tagged example-voting-app_vote:latest
WARNING: Image for service vote was built because it did not
already exist. To rebuild this image you must use `docker-
compose build` or `docker-compose up --build`.
```

Once Docker Compose has built the `vote` image, it starts on building the `result` image, as follows:

```
Building result
Step 1/9 : FROM node:10-slim
10-slim: Pulling from library/node
619014d83c02: Pull complete
8c5d9aed65fb: Pull complete
ec6ca7c6739a: Pull complete
6da8fc40e075: Pull complete
6161f60894b2: Pull complete
Digest: sha256:10c4d19a2a2fa5ad416bddb3a4b208e
34b0d4263c3978df6aa06d9ba9687bbe8
Status: Downloaded newer image for node:10-slim
 ---> ad4ea09bf0f3
Again, now the images have been downloaded the build of the
image containing the application can start:
Step 2/9 : WORKDIR /app
 ---> Running in 040efda3a918
Removing intermediate container 040efda3a918
 ---> 3d3326950331
Step 3/9 : RUN npm install -g nodemon
 ---> Running in a0ce3043aba5
[lots of nodejs build output here]
Removing intermediate container a0ce3043aba5
 ---> 925a30942e5f
Step 4/9 : COPY package*.json ./
 ---> 9fd59fddc0e8
Step 5/9 : RUN npm ci  && npm cache clean --force  && mv /app/
node_modules /node_modules
 ---> Running in 3c0871538d04
[lots of nodejs build output here]
```

```
Removing intermediate container 3c0871538d04
 ---> 8db74baa1959
Step 6/9 : COPY . .
 ---> a47af934177b
Step 7/9 : ENV PORT 80
 ---> Running in 57f80f86faf0
Removing intermediate container 57f80f86faf0
 ---> e5a01939876b
Step 8/9 : EXPOSE 80
 ---> Running in 614bd7bd4ab3
Removing intermediate container 614bd7bd4ab3
 ---> 461355b7e66e
Step 9/9 : CMD ["node", "server.js"]
 ---> Running in 4c64da5f054c
Removing intermediate container 4c64da5f054c
 ---> 65c854a0b292
Successfully built 65c854a0b292
Successfully tagged example-voting-app_result:latest
WARNING: Image for service result was built because it did
not already exist. To rebuild this image you must use `docker-
compose build` or `docker-compose up --build`.
```

This is followed by the PostgreSQL image for the `db` container being pulled down from Docker Hub, as illustrated here:

```
Pulling db (postgres:9.4)...
9.4: Pulling from library/postgres
619014d83c02: Already exists
7ec0fe6664f6: Pull complete
9ca7ba8f7764: Pull complete
9e1155d037e2: Pull complete
febcfb7f8870: Pull complete
8c78c79412b5: Pull complete
5a35744405c5: Pull complete
27717922e067: Pull complete
8e8ebde0a697: Pull complete
f6d85e336541: Pull complete
c802081bbe1e: Pull complete
```

```
f35abd4ea98b: Pull complete
50335e437328: Pull complete
a1c34d9ddebb: Pull complete
Digest: sha256:d6bc1739199cc52f038f54e1ab
671f5229d114fb667e9ad08add6cd66e8a9b28
Status: Downloaded newer image for postgres:9.4
```

Finally, the `worker` image is constructed, as follows:

```
Building worker
Step 1/5 : FROM microsoft/dotnet:2.0.0-sdk
2.0.0-sdk: Pulling from microsoft/dotnet
3e17c6eae66c: Pull complete
74d44b20f851: Pull complete
a156217f3fa4: Pull complete
4a1ed13b6faa: Pull complete
18842ff6b0bf: Pull complete
e857bd06f538: Pull complete
b800e4c6f9e9: Pull complete
Digest: sha256:f4ea9cdf980bb9512523a3fb88e
30f2b83cce4b0cddd2972bc36685461081e2f
Status: Downloaded newer image for microsoft/dotnet:2.0.0-sdk
Now that the SDK images have been downloaded Docker Compose can
build the application:
 ---> fde8197d13f4
Step 2/5 : WORKDIR /code
 ---> Running in ac782e4c8cb2
Removing intermediate container ac782e4c8cb2
 ---> 3881e09f0d22
Step 3/5 : ADD src/Worker /code/src/Worker
 ---> cf0468608709
Step 4/5 : RUN dotnet restore -v minimal src/Worker     &&
dotnet publish -c Release -o "./" "src/Worker/"
 ---> Running in ca04867b0e86
[lots of .net build output here]
  Worker -> /code/src/Worker/bin/Release/netcoreapp2.0/Worker.
dll
  Worker -> /code/src/Worker/
```

```
Removing intermediate container ca04867b0e86
 ---> 190aee9b4b98
Step 5/5 : CMD dotnet src/Worker/Worker.dll
 ---> Running in 069b5806b25e
Removing intermediate container 069b5806b25e
 ---> 56c488a158bb
Successfully built 56c488a158bb
Successfully tagged example-voting-app_worker:latest
WARNING: Image for service worker was built because it did
not already exist. To rebuild this image you must use `docker-
compose build` or `docker-compose up --build`.
```

You may have noticed that the Redis image being used by the redis container was not pulled—this is because the latest version was already downloaded. Now that all the images have either been built or pulled down and the networking and volumes are in place, Docker Compose can launch our application, as follows:

```
Creating redis                          ... done
Creating db                             ... done
Creating example-voting-app_vote_1      ... done
Creating example-voting-app_result_1 ... done
Creating example-voting-app_worker_1 ... done
Attaching to db, redis, example-voting-app_worker_1, example-
voting-app_vote_1, example-voting-app_result_1
```

The result part of the application can be accessed at http://localhost:5001. By default, there are no votes and it is split 50/50, as illustrated in the following screenshot:

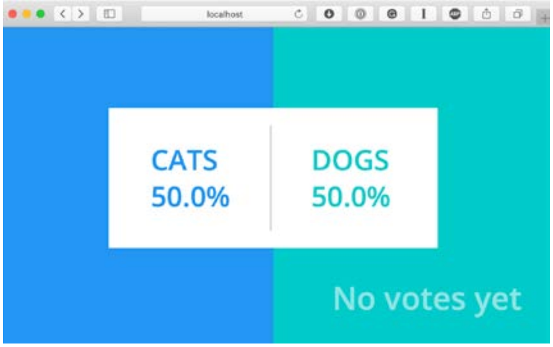


Figure 5.4 – The default voting tally

The `voting` part of the application can be found at `http://localhost:5000`, and is shown in the following screenshot:

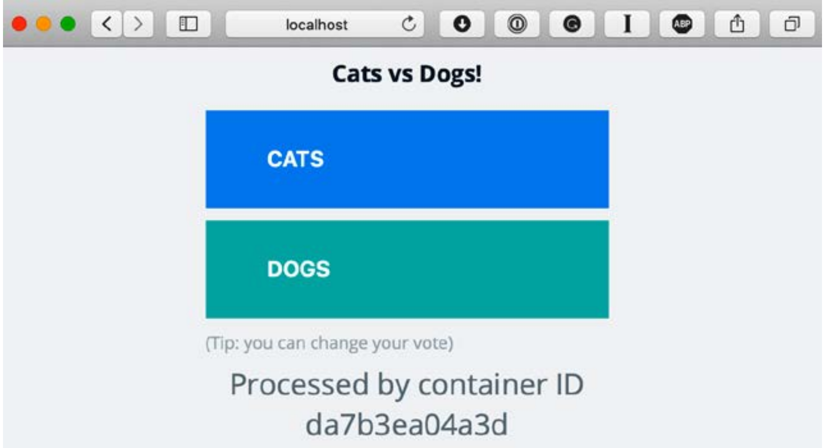
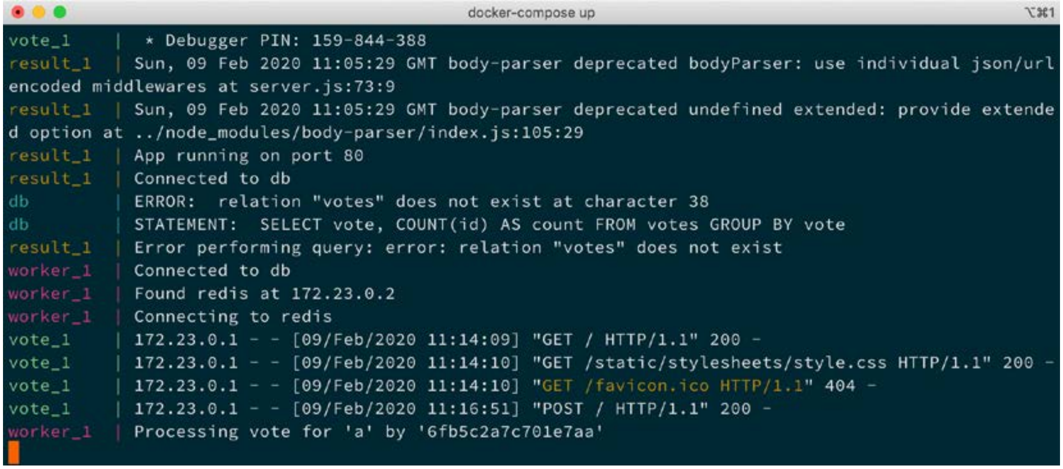


Figure 5.5 – The voting interface

Clicking on either **CATS** or **DOGS** will register a vote; you should be able to see this logged in the Docker Compose output in your Terminal, as follows:



```
vote_1      |  * Debugger PIN: 159-844-388
result_1    | Sun, 09 Feb 2020 11:05:29 GMT body-parser deprecated bodyParser: use individual json/url
encoded middlewares at server.js:73:9
result_1    | Sun, 09 Feb 2020 11:05:29 GMT body-parser deprecated undefined extended: provide extende
d option at ../node_modules/body-parser/index.js:105:29
result_1    | App running on port 80
result_1    | Connected to db
db          | ERROR:  relation "votes" does not exist at character 38
db          | STATEMENT:  SELECT vote, COUNT(id) AS count FROM votes GROUP BY vote
result_1    | Error performing query: error: relation "votes" does not exist
worker_1    | Connected to db
worker_1    | Found redis at 172.23.0.2
worker_1    | Connecting to redis
vote_1      | 172.23.0.1 - - [09/Feb/2020 11:14:09] "GET / HTTP/1.1" 200 -
vote_1      | 172.23.0.1 - - [09/Feb/2020 11:14:10] "GET /static/stylesheets/style.css HTTP/1.1" 200 -
vote_1      | 172.23.0.1 - - [09/Feb/2020 11:14:10] "GET /favicon.ico HTTP/1.1" 404 -
vote_1      | 172.23.0.1 - - [09/Feb/2020 11:16:51] "POST / HTTP/1.1" 200 -
worker_1    | Processing vote for 'a' by '6fb5c2a7c701e7aa'
```

Figure 5.6 – docker-compute output showing votes

There are a few errors, as the Redis table structure is only created when the vote application registers the first vote; once a vote has been cast, the Redis table structure will be created and the `worker` container will take that vote and process it by writing to the `db` container. Once the vote has been cast, the `result` container will update in real time, as illustrated in the following screenshot:

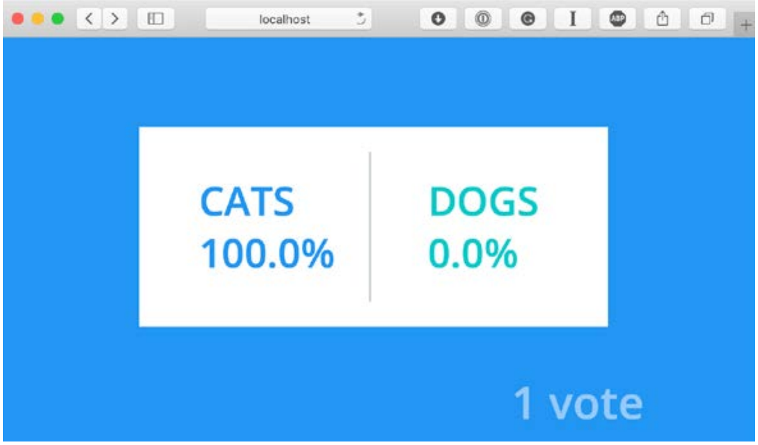
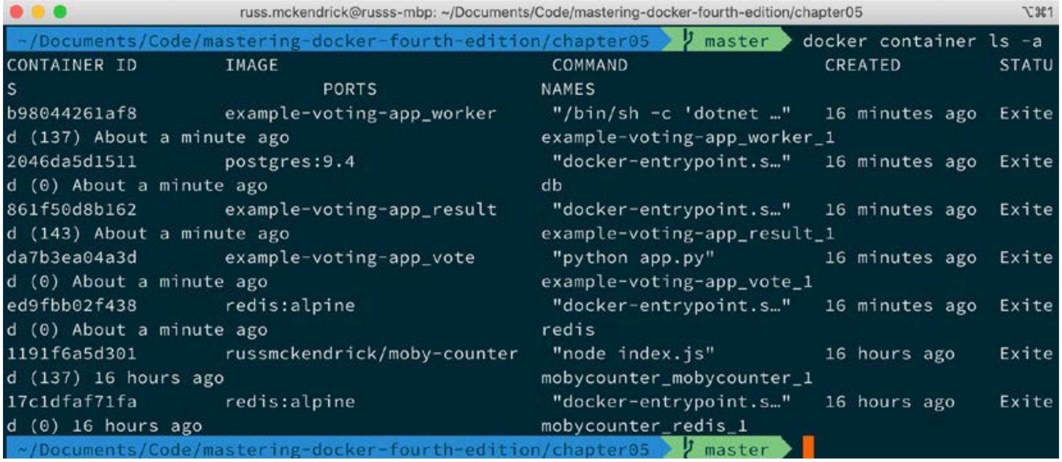


Figure 5.7 – Vote result page after vote was cast

We will be looking at the Docker Compose YAML files again in the upcoming chapters, when we look at launching both Docker Swarm stacks and Kubernetes clusters. For now, let's get back to Docker Compose and look at some of the commands we can run.

Exploring Docker Compose commands

We are over halfway through the chapter, and the only Docker Compose command we have run is `docker-compose up`. If you have been following along and you run `docker container ls -a`, you will see something similar to the following Terminal screen:


russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05
⌥⌘1

```
~/Documents/Code/mastering-docker-fourth-edition/chapter05  master  docker container ls -a
CONTAINER ID        IMAGE                         COMMAND                   CREATED         STATU
S                            PORTS               NAMES
b98044261af8        example-voting-app_worker     "/bin/sh -c 'dotnet …"    16 minutes ago  Exite
d (137) About a minute ago                       example-voting-app_worker_1
2046da5d1511        postgres:9.4                  "docker-entrypoint.s…"    16 minutes ago  Exite
d (0) About a minute ago                         db
861f50d8b162        example-voting-app_result     "docker-entrypoint.s…"    16 minutes ago  Exite
d (143) About a minute ago                       example-voting-app_result_1
da7b3ea04a3d        example-voting-app_vote       "python app.py"           16 minutes ago  Exite
d (0) About a minute ago                         example-voting-app_vote_1
ed9fbb02f438        redis:alpine                  "docker-entrypoint.s…"    16 minutes ago  Exite
d (0) About a minute ago                         redis
1191f6a5d301        russmckendrick/moby-counter   "node index.js"           16 hours ago    Exite
d (137) 16 hours ago                             mobycounter_mobycounter_1
17c1dfaf71fa        redis:alpine                  "docker-entrypoint.s…"    16 hours ago    Exite
d (0) 16 hours ago                               mobycounter_redis_1
~/Documents/Code/mastering-docker-fourth-edition/chapter05  master 
```

Figure 5.8 – Output for docker container ls -a

As you can see, we have a lot of containers with the status of `Exited`. This is because when we used *Ctrl* + *C* to return to our Terminal, the Docker Compose containers were stopped.

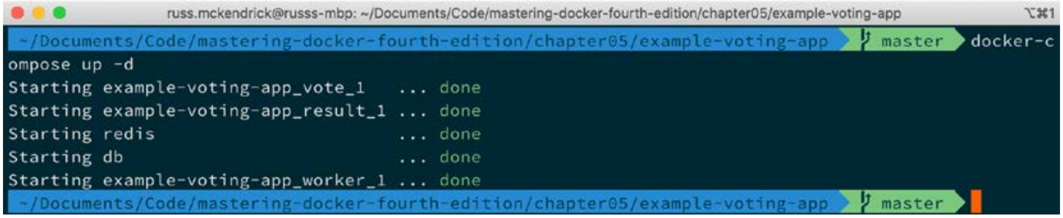
Choose one of the Docker Compose applications and change to the folder that contains the `docker-compose.yml` file, and we will work through some more Docker Compose commands. I will be using the **Example Vote** application.

up and ps

The first command is `docker-compose up`, but this time, we will be adding a flag. In your chosen application folder, run the following:

```
$ docker-compose up -d
```

This will start your application back up, this time in detached mode, which will return you to your Terminal prompt, as follows:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app ⌥⌘1

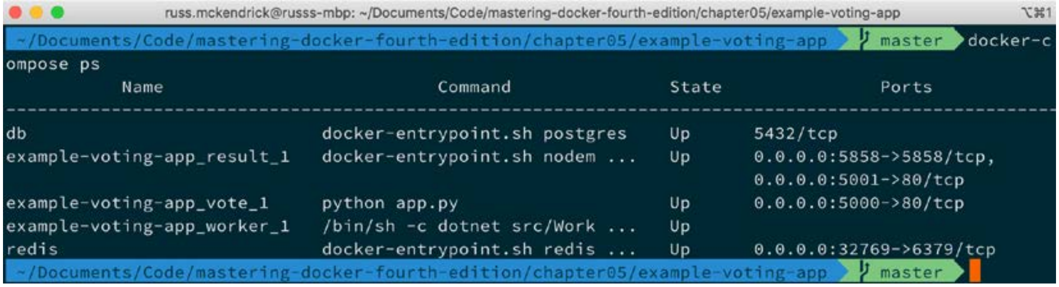
```
 ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master  docker-c
ompose up -d
Starting example-voting-app_vote_1   ... done
Starting example-voting-app_result_1 ... done
Starting redis                       ... done
Starting db                          ... done
Starting example-voting-app_worker_1 ... done
 ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master 
```

Figure 5.9 – Output for docker-compose up -d

Once control of your Terminal is returned to you, you should be able to check that the containers are running, using the following command:

```
$ docker-compose ps
```

As you can see from the following Terminal output, all of the containers have the state of `Up`:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app ⌥⌘1

```
 ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master  docker-c
ompose ps
           Name                          Command               State              Ports
------------------------------------------------------------------------------------------------------
db                            docker-entrypoint.sh postgres   Up      5432/tcp
example-voting-app_result_1   docker-entrypoint.sh nodem ...   Up      0.0.0.0:5858->5858/tcp,
                                                                      0.0.0.0:5001->80/tcp
example-voting-app_vote_1     python app.py                   Up      0.0.0.0:5000->80/tcp
example-voting-app_worker_1   /bin/sh -c dotnet src/Work ...   Up
redis                         docker-entrypoint.sh redis ...   Up      0.0.0.0:32769->6379/tcp
 ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master 
```

Figure 5.10 – Output for docker-compose ps

When running these commands, Docker Compose will only be aware of the containers defined in the `services` section of your `docker-compose.yml` file; all other containers will be ignored as they don't belong to our service stack.

config

Running the following command will validate our `docker-compose.yml` file:

```
$ docker-compose config
```

If there are no issues, it will print a rendered copy of your Docker Compose YAML file to screen; this is how Docker Compose will interpret your file. If you don't want to see this output and just want to check for errors, then you can run the following command:

```
$ docker-compose config -q
```

This is shorthand for `--quiet`. If there are any errors (which the examples we have worked through so far shouldn't have), they will be displayed as follows:

```
ERROR: yaml.parser.ParserError: while parsing a block mapping
in "./docker- compose.yml", line 1, column 1 expected <block
end>, but found '<block mapping start>' in "./docker-compose.
yml", line 27, column 3
```

pull, build, and create

The next two commands will help you prepare to launch your Docker Compose application. The following command will read your Docker Compose YAML file and pull any of the images it finds:

```
$ docker-compose pull
```

The following command will execute any build instructions it finds in your file:

```
$ docker-compose build
```

These commands are useful when you are first defining your Docker Compose-powered application and want to test it without launching your application. The `docker-compose build` command can also be used to trigger a build if there are updates to any of the Dockerfiles used to originally build your images.

The pull and build commands only generate/pull the images needed for our application; they do not configure the containers themselves. For this, we need to use the following command:

```
$ docker-compose create
```

This will create but not launch the containers. As with the docker container create command, they will have an Exited state until you start them. The create command has a few useful flags you can pass, as follows:

- --force-recreate: This recreates the container even if there is no need to, as nothing within the configuration has changed.
- --no-recreate: This doesn't recreate a container if it already exists; this flag cannot be used with the preceding flag.
- --no-build: This doesn't build the images, even if an image that needs to be built is missing.
- --build: This builds the images before creating the containers.

start, stop, restart, pause, and unpause

The following commands work in exactly the same way as their docker container counterparts, the only difference being that they effect change on all of the containers:

```
$ docker-compose start
$ docker-compose stop
$ docker-compose restart
$ docker-compose pause
$ docker-compose unpause
```

It is possible to target a single service by passing its name; for example, to pause and unpause the db service, we would run the following:

```
$ docker-compose pause db
$ docker-compose unpause db
```

Now that we know how to stop and start all or part of our Docker Compose application, we can look at how we can see some more information on our application.

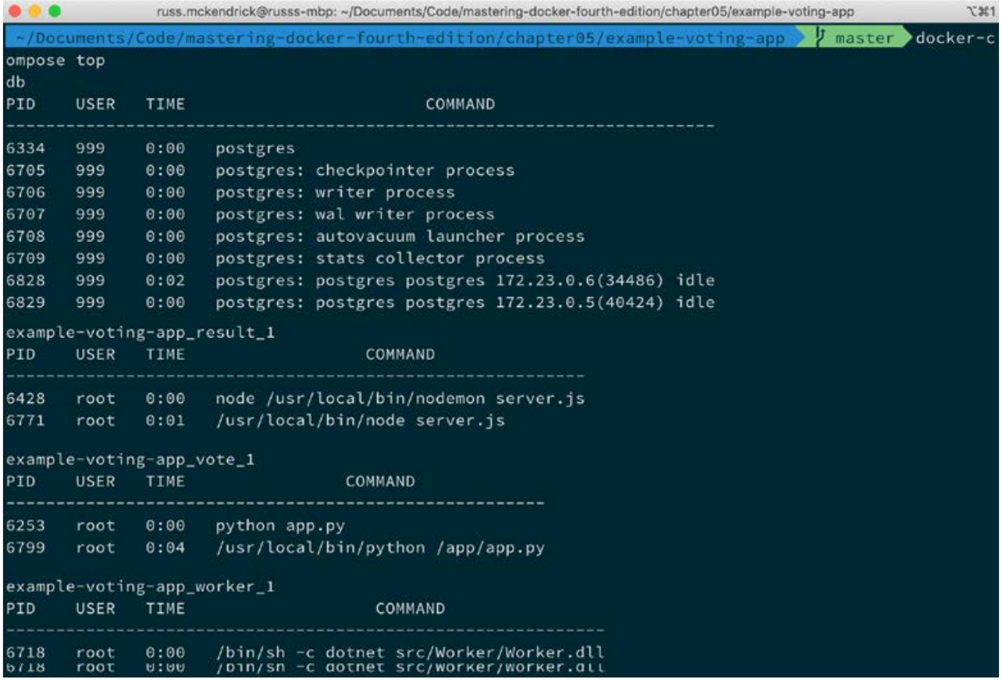
top, logs, events, exec, and run

The next three commands all give us feedback on what is happening within our running containers and Docker Compose.

The following command, as with its `docker container` counterpart, displays information on the processes running within each of our Docker Compose-launched containers:

```
$ docker-compose top
```

As you can see from the following Terminal output, each container is split into its own section:


russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app
⌥⌘1

```
 ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master docker-c
ompose top
db
PID    USER   TIME                        COMMAND
-----------------------------------------------------------------------
6334   999    0:00   postgres
6705   999    0:00   postgres: checkpointer process
6706   999    0:00   postgres: writer process
6707   999    0:00   postgres: wal writer process
6708   999    0:00   postgres: autovacuum launcher process
6709   999    0:00   postgres: stats collector process
6828   999    0:02   postgres: postgres postgres 172.23.0.6(34486) idle
6829   999    0:00   postgres: postgres postgres 172.23.0.5(40424) idle

example-voting-app_result_1
PID    USER   TIME                  COMMAND
----------------------------------------------------------
6428   root   0:00   node /usr/local/bin/nodemon server.js
6771   root   0:01   /usr/local/bin/node server.js

example-voting-app_vote_1
PID    USER   TIME                COMMAND
------------------------------------------------------
6253   root   0:00   python app.py
6799   root   0:04   /usr/local/bin/python /app/app.py

example-voting-app_worker_1
PID    USER   TIME                   COMMAND
------------------------------------------------------------
6718   root   0:00   /bin/sh -c dotnet src/Worker/Worker.dll
6718   root   0:00   /bin/sh -c dotnet src/Worker/Worker.dll
```

Figure 5.11 – Output for docker-compose top

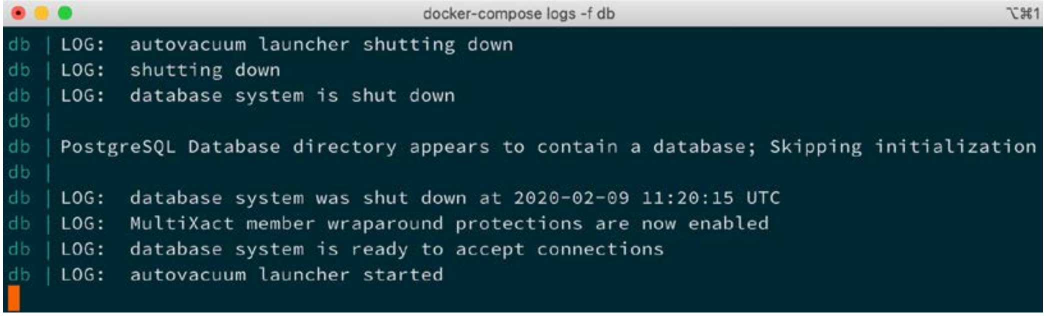
If you would just like to see one of the services, you simply have to pass its name when running the command, as follows:

```
$ docker-compose top db
```

The following command streams the logs from each of the running containers to screen:

```
$ docker-compose logs
```

As with the `docker container` command, you can pass flags such as `-f` or `--follow` to keep the stream flowing until you press *Ctrl* + *C*. Also, you can stream the logs for a single service by appending its name to the end of your command, as follows:



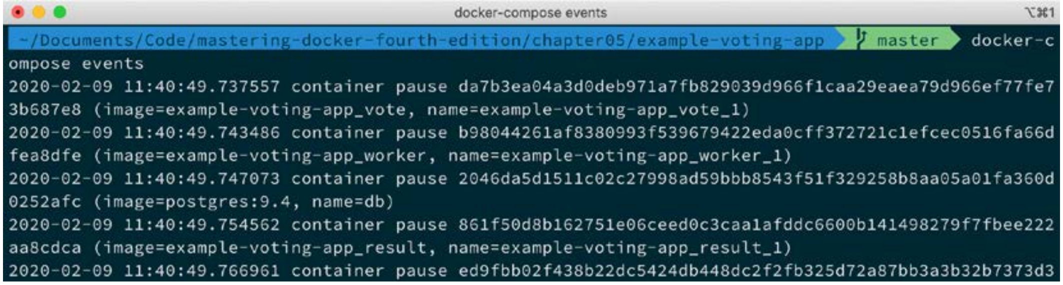
```
db | LOG:  autovacuum launcher shutting down
db | LOG:  shutting down
db | LOG:  database system is shut down
db |
db | PostgreSQL Database directory appears to contain a database; Skipping initialization
db |
db | LOG:  database system was shut down at 2020-02-09 11:20:15 UTC
db | LOG:  MultiXact member wraparound protections are now enabled
db | LOG:  database system is ready to accept connections
db | LOG:  autovacuum launcher started
```

Figure 5.12 – Log stream

The `events` command again works like the `docker container` equivalent; it streams events—such as the ones triggered by the other commands we have been discussing—in real time. For example, run this command:

```
$ docker-compose events
```

Running `docker-compose pause` in a second Terminal window gives the following output:



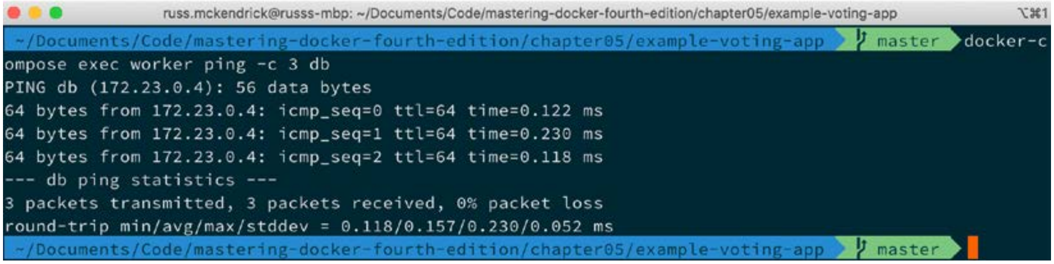
```
~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master  docker-c
ompose events
2020-02-09 11:40:49.737557 container pause da7b3ea04a3d0deb971a7fb829039d966f1caa29eaea79d966ef77fe7
3b687e8 (image=example-voting-app_vote, name=example-voting-app_vote_1)
2020-02-09 11:40:49.743486 container pause b98044261af8380993f539679422eda0cff372721c1efcec0516fa66d
fea8dfe (image=example-voting-app_worker, name=example-voting-app_worker_1)
2020-02-09 11:40:49.747073 container pause 2046da5d1511c02c27998ad59bbb8543f51f329258b8aa05a01fa360d
0252afc (image=postgres:9.4, name=db)
2020-02-09 11:40:49.754562 container pause 861f50d8b162751e06ceed0c3caa1afddc6600b141498279f7fbee222
aa8cdca (image=example-voting-app_result, name=example-voting-app_result_1)
2020-02-09 11:40:49.766961 container pause ed9fbb02f438b22dc5424db448dc2f2fb325d72a87bb3a3b32b7373d3
```

Figure 5.13 – Output for docker-compose events

These two commands run similarly to their `docker container` equivalents. Run the following:

```
$ docker-compose exec worker ping -c 3 db
```

This will launch a new process in the already running worker container and ping the `db` container three times, as seen here:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app   ⌥⌘1
~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master  docker-c
ompose exec worker ping -c 3 db
PING db (172.23.0.4): 56 data bytes
64 bytes from 172.23.0.4: icmp_seq=0 ttl=64 time=0.122 ms
64 bytes from 172.23.0.4: icmp_seq=1 ttl=64 time=0.230 ms
64 bytes from 172.23.0.4: icmp_seq=2 ttl=64 time=0.118 ms
--- db ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.118/0.157/0.230/0.052 ms
~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master
```

Figure 5.14 – Output for docker-compose exec worker ping -c 3 db

The `run` command is useful if you need to run a containerized command as a one-off within your application. For example, if you use a package manager such as `composer` to update the dependencies of your project that is stored on a volume, you could run something like this:

```
$ docker-compose run --volume data_volume:/app composer install
```

This would run the `composer` container with the `install` command and mount the `data_volume` at the following path `/app` within the container.

scale

The `scale` command will take the service you pass to the command and scale it to the number you define. For example, to add more `worker` containers, I just need to run the following:

```
$ docker-compose scale worker=3
```

However, this actually gives the following warning:

```
WARNING: The scale command is deprecated. Use the up command
with the - scale flag instead.
```

What we should now be using is the following command:

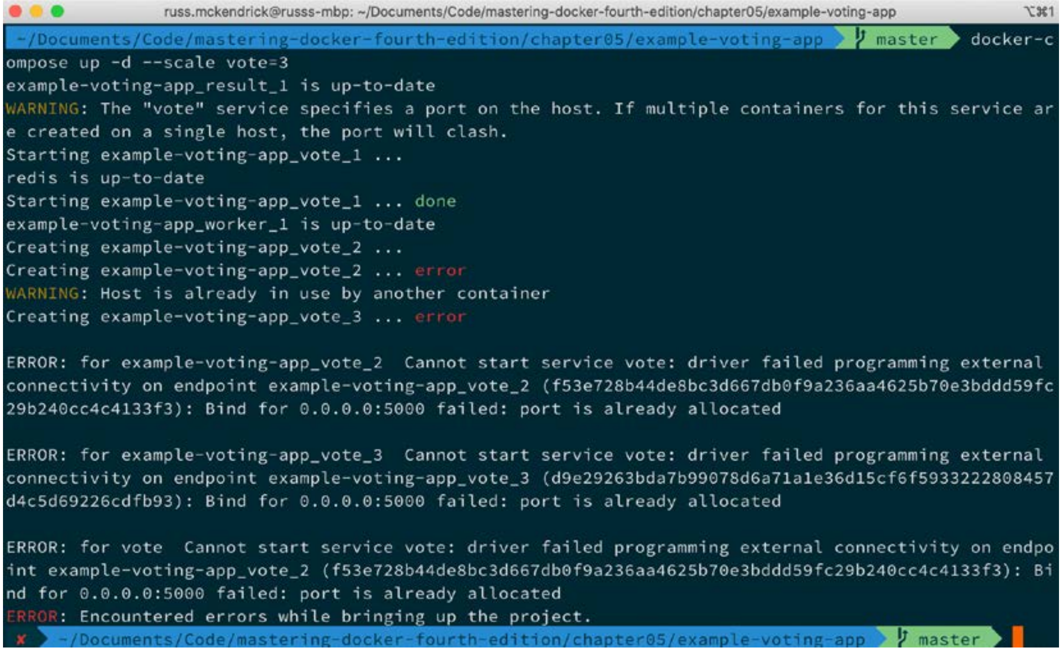
```
$ docker-compose up -d --scale worker=3
```

While the `scale` command is in the current version of Docker Compose, it will be removed from future versions of the software.

You will notice that I chose to scale the number of `worker` containers. There is a good reason for this, as you will see for yourself if you try running the following command:

```
$ docker-compose up -d --scale vote=3
```

You will notice that while Docker Compose creates the additional two containers, they fail to start, with the following error:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app ⌥⌘1

```
~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master  docker-c
ompose up -d --scale vote=3
example-voting-app_result_1 is up-to-date
WARNING: The "vote" service specifies a port on the host. If multiple containers for this service ar
e created on a single host, the port will clash.
Starting example-voting-app_vote_1 ...
redis is up-to-date
Starting example-voting-app_vote_1 ... done
example-voting-app_worker_1 is up-to-date
Creating example-voting-app_vote_2 ...
Creating example-voting-app_vote_2 ... error
WARNING: Host is already in use by another container
Creating example-voting-app_vote_3 ... error

ERROR: for example-voting-app_vote_2  Cannot start service vote: driver failed programming external
connectivity on endpoint example-voting-app_vote_2 (f53e728b44de8bc3d667db0f9a236aa4625b70e3bddd59fc
29b240cc4c4133f3): Bind for 0.0.0.0:5000 failed: port is already allocated

ERROR: for example-voting-app_vote_3  Cannot start service vote: driver failed programming external
connectivity on endpoint example-voting-app_vote_3 (d9e29263bda7b99078d6a71a1e36d15cf6f593322808457
d4c5d69226cdfb93): Bind for 0.0.0.0:5000 failed: port is already allocated

ERROR: for vote  Cannot start service vote: driver failed programming external connectivity on endpo
int example-voting-app_vote_2 (f53e728b44de8bc3d667db0f9a236aa4625b70e3bddd59fc29b240cc4c4133f3): Bi
nd for 0.0.0.0:5000 failed: port is already allocated
ERROR: Encountered errors while bringing up the project.
 ✘  ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master 
```

Figure 5.15 – Output for docker-compose up -d --scale vote=3

That is because we cannot have three individual containers all trying to map to the same port on the host machine; because of this, you should always use the `scale` command on containers where you haven't explicitly defined a port mapping.

kill, rm, and down

The three Docker Compose commands we are going to look at last are the ones that remove/terminate our Docker Compose application. The first command stops our running containers by immediately stopping running container processes. This is the `kill` command, as illustrated here:

```
$ docker-compose kill
```

Be careful when running this as it does not wait for containers to gracefully stop, such as when running `docker-compose stop`, meaning that using the `docker-compose kill` command may result in data loss.

Next up is the `rm` command; this removes any containers with the state of `exited`, and is shown here:

```
$ docker-compose rm
```

Finally, we have the `down` command. This, as you might have already guessed, has the opposite effect of running `docker-compose up`, and is shown here:

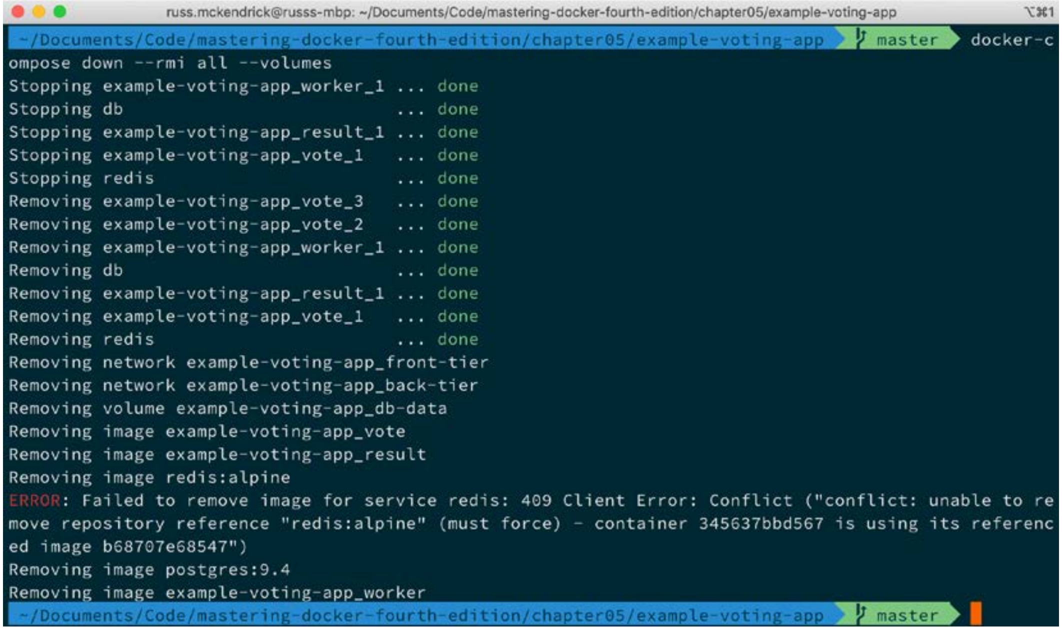
```
$ docker-compose down
```

That will remove the containers and the networks created when running `docker-compose up`. If you want to remove everything, you can do so by running the following:

```
$ docker-compose down --rmi all --volumes
```

This will remove all of the containers, networks, volumes, and images (both pulled and built) when you ran the `docker-compose up` command; this includes images that may be in use outside of your Docker Compose application.

There will, however, be an error if the images are in use, and those images will not be removed, as illustrated in the following screenshot:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app
⌥⌘1

```
~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master  docker-compose down --rmi all --volumes
Stopping example-voting-app_worker_1 ... done
Stopping db                          ... done
Stopping example-voting-app_result_1 ... done
Stopping example-voting-app_vote_1   ... done
Stopping redis                       ... done
Removing example-voting-app_vote_3   ... done
Removing example-voting-app_vote_2   ... done
Removing example-voting-app_worker_1 ... done
Removing db                          ... done
Removing example-voting-app_result_1 ... done
Removing example-voting-app_vote_1   ... done
Removing redis                       ... done
Removing network example-voting-app_front-tier
Removing network example-voting-app_back-tier
Removing volume example-voting-app_db-data
Removing image example-voting-app_vote
Removing image example-voting-app_result
Removing image redis:alpine
ERROR: Failed to remove image for service redis: 409 Client Error: Conflict ("conflict: unable to remove repository reference "redis:alpine" (must force) - container 345637bbd567 is using its referenced image b68707e68547")
Removing image postgres:9.4
Removing image example-voting-app_worker
~/Documents/Code/mastering-docker-fourth-edition/chapter05/example-voting-app  master
```

Figure 5.16 – Output of docker-compose down --rmi all --volumes

As you can see from the preceding output, there is a container using the `redis` image, the Moby counter application, so it was not removed. However, all other images used by the `Example Vote` application are removed, both the ones built as part of the initial `docker- compose up` command and the ones downloaded from Docker Hub.

Using Docker App

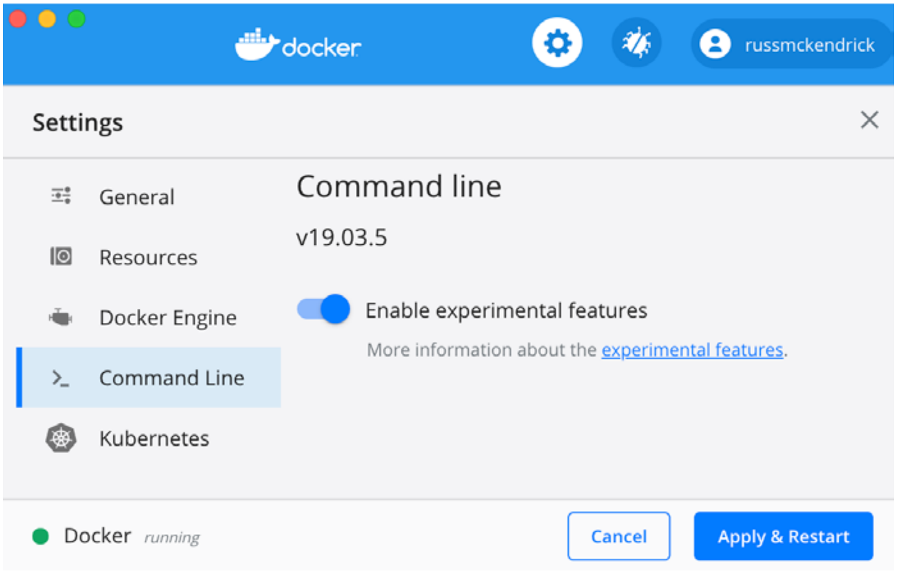
Before we start this section, I should issue the following warning:

> **Important note**
>
> The feature we are going to discuss is very much an experimental one. It is in its very early stages of development and should not be considered any more than a preview of an upcoming feature.

Because of its experimental nature, I am only going to cover the usage of Docker App on macOS. However, before we enable it, let's discuss what exactly is meant by a Docker app. While Docker Compose files are really useful when it comes to sharing your environment with others, you may have noticed that there is one quite crucial element we have been missing so far in this chapter, and that is the ability to actually distribute your Docker Compose files in a similar way to how you can distribute your Docker images.

Docker has acknowledged this and is currently working on a new feature called Docker App, which it hopes will fill this gap.

Docker App is currently a command-line client plugin that helps you to create an application bundle that can be shared via Docker Hub or a Docker Enterprise Registry. The plugin is built in to Docker 19.03, and all you have to do is open the Docker desktop **Settings** and toggle on **Enable experimental features**, as illustrated in the following screenshot:



Command line

v19.03.5

Enable experimental features

More information about the experimental features.

Docker running

Figure 5.17 – Docker desktop Enable experimental features screen

We are going to be using the Moby Counter application as its `docker-compose.yml` file already meets the pre-requisites for a Docker App bundle, as we are using a version of 3.6 and higher—3.7, in our case.

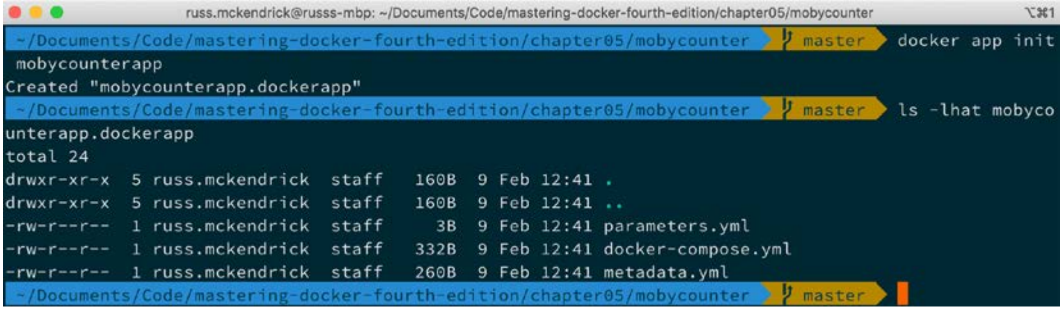
Defining the application

Let's begin by following these steps:

1. We need to create our Docker App configuration. To do this, change to the `mobycounter` folder and run the following command:

```
$ docker app init mobycounterapp
```

This will create a folder called `mobycounterapp.dockerapp`. In that folder, there are three files, as can be seen in the following screenshot:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter    ⌥⌘1
~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master  docker app init
 mobycounterapp
Created "mobycounterapp.dockerapp"
~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master  ls -lhat mobyco
unterapp.dockerapp
total 24
drwxr-xr-x  5 russ.mckendrick  staff   160B  9 Feb 12:41 .
drwxr-xr-x  5 russ.mckendrick  staff   160B  9 Feb 12:41 ..
-rw-r--r--  1 russ.mckendrick  staff     3B  9 Feb 12:41 parameters.yml
-rw-r--r--  1 russ.mckendrick  staff   332B  9 Feb 12:41 docker-compose.yml
-rw-r--r--  1 russ.mckendrick  staff   260B  9 Feb 12:41 metadata.yml
~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master
```

Figure 5.18 – Output of ls -lhat mobycounterapp.dockerapp

The `docker-compose.yml` file is currently a copy of our original file; the `parameters.yml` file doesn't contain any data; and the `metadata.yml` file, for me, currently looks like the following:

```
# Version of the application
version: 0.1.0
# Name of the application
name: mobycounterapp
# A short description of the application
description:
# List of application maintainers with name and email for each
maintainers:
  - name: russ.mckendrick
    email:
```

2. Let's start by updating the `metadata.yml` file. I updated mine to read as follows:

```
# Version of the application
version: 0.1.0
```

```
# Name of the application
name: mobycounterapp
# A short description of the application
description: Places whales on screen wherever you click
!!!
# List of application maintainers with name and email for
each
maintainers:
  - name: Russ McKendrick
    email: russ@mckendrick.io
```

The preceding information will be used when we distribute the application. Initially, this will be via the Docker Hub, so please make sure that you are happy for the information to be accessible to other people.

3. Now that we have our metadata, let's add some parameters to the `parameters.yml` file, as follows:

```
{
    "port": "8080"
}
```

4. Finally, update the `docker-compose.yml` file to make use of the parameters we have just defined, as follows:

```
version: "3.7"

services:
    redis:
        image: redis:alpine
        volumes:
           - redis_data:/data
        restart: always
    mobycounter:
        depends_on:
            - redis
        image: russmckendrick/moby-counter
        ports:
            - "${port}:80"
        restart: always
```

```

volumes:
    redis_data:
```

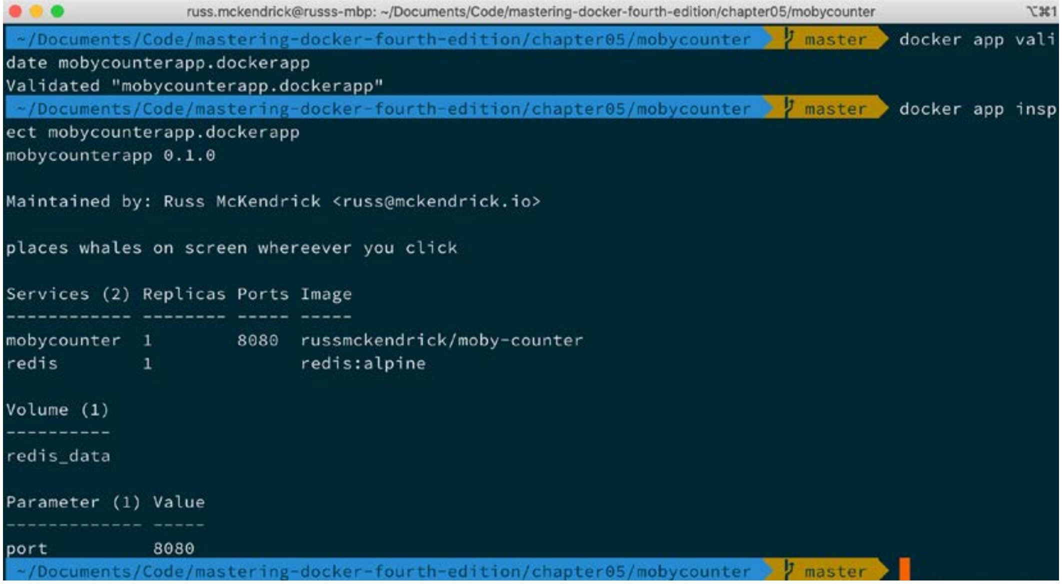
As you can see, I have added `${port}` to the `docker-compose.yml` file. When we launch the application, the values will be populated from the `parameters.yml` file.

Validating and inspecting the application

We can double-check that everything we have changed is OK by running the following commands:

```
$ docker app validate mobycounterapp.dockerapp
$ docker app inspect mobycounterapp.dockerapp
```

If everything is OK, this should display something that looks like the following Terminal output:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter
⌥⌘1

```
~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master  docker app vali
date mobycounterapp.dockerapp
Validated "mobycounterapp.dockerapp"
~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master  docker app insp
ect mobycounterapp.dockerapp
mobycounterapp 0.1.0

Maintained by: Russ McKendrick <russ@mckendrick.io>

places whales on screen whereever you click

Services (2) Replicas Ports Image
------------ -------- ----- -----
mobycounter  1        8080  russmckendrick/moby-counter
redis        1              redis:alpine

Volume (1)
----------
redis_data

Parameter (1) Value
------------- -----
port          8080
~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master 
```

Figure 5.19 – Output for docker app inspect mobycounterapp.dockerapp

Once we've validated the app, we can finally move on to launching it.

Launching the app

For us to be able to launch the application natively, we need to be running a Docker Swarm cluster (we will be covering Docker Swarm in more detail in a future chapter). To achieve this, proceed as follows:

1. Begin by running the following command:

   ```
   $ docker swarm init
   ```

   Ignore the output for now—we won't be creating a cluster just yet and only need a single node.

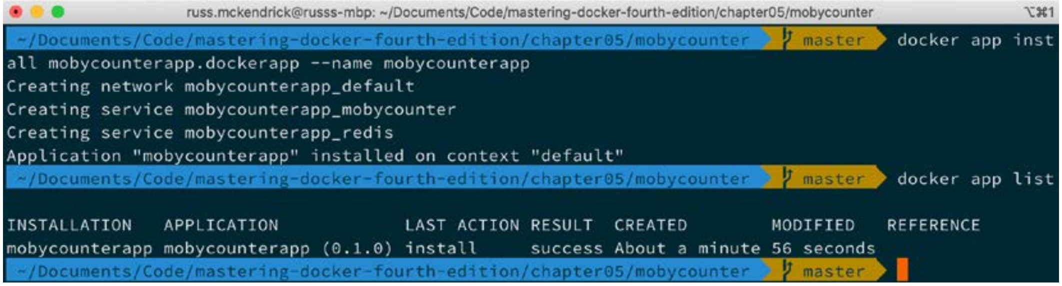
2. Now that we have Docker Swarm enabled, we can install the application using the following command:

   ```
   $ docker app install mobycounterapp.dockerapp --name
   mobycounterapp
   ```

3. Once installed, you can run the following command to check the status of the application:

   ```
   $ docker app list
   ```

You should see something like the following Terminal output:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter          ⌥⌘1
 ~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master  docker app inst
all mobycounterapp.dockerapp --name mobycounterapp
Creating network mobycounterapp_default
Creating service mobycounterapp_mobycounter
Creating service mobycounterapp_redis
Application "mobycounterapp" installed on context "default"
 ~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master  docker app list

INSTALLATION   APPLICATION            LAST ACTION RESULT  CREATED        MODIFIED   REFERENCE
mobycounterapp mobycounterapp (0.1.0) install     success About a minute 56 seconds
 ~/Documents/Code/mastering-docker-fourth-edition/chapter05/mobycounter  master 
```

Figure 5.20 – Output for docker app install and docker app list

The application should also be accessible on `http://localhost:8080`.

Pushing to Docker Hub

While our application is running, we can publish it to the Docker Hub. To do this, proceed as follows:

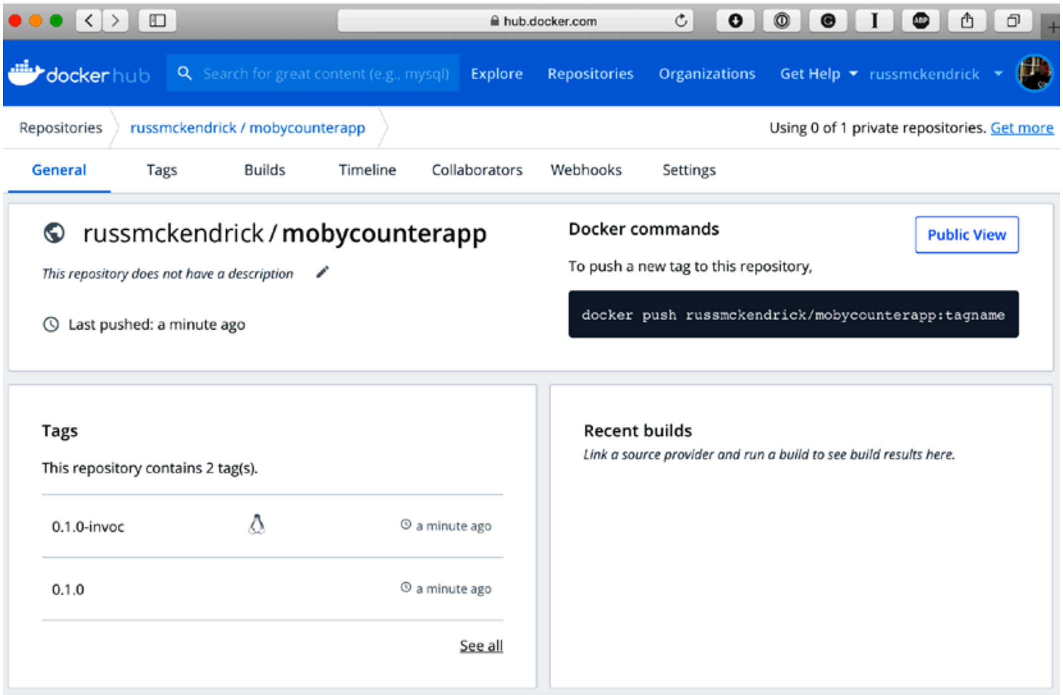
1. If you haven't done so already, log in by running the following command:

```
$ docker login
```

2. Once logged in, run the following command, making sure that you update the Docker Hub ID (which is russmckendrick in my case) with your own:

```
$ docker app push mobycounterapp --platform="linux/amd64"
--tag russmckendrick/mobycounterapp:0.1.0
```

After a minute, you should see a message that the bundle has been successfully pushed, as illustrated in the following screenshot:



russmckendrick / mobycounterapp

This repository does not have a description

Last pushed: a minute ago

Docker commands

To push a new tag to this repository,

```
docker push russmckendrick/mobycounterapp:tagname
```

Tags

This repository contains 2 tag(s).

0.1.0-invoc	a minute ago
0.1.0	a minute ago

Recent builds

Link a source provider and run a build to see build results here.

Figure 5.21 – hub.docker.com page

Opening `https://hub.docker.com/` in your browser should show that the application is there, as seen in *Figure 5.21*.

Installing from Docker Hub

First of all, let's create a temporary folder and uninstall the application, as follows:

```
$ mkdir /tmp/testing
$ cd /tmp/testing
$ docker app uninstall mobycounterapp
$ docker app ls
```

As you can see from the output if you are following along, we don't have any applications running, and we have changed folders away from our original `docker-compose.yml` file and also the `mobycounterapp.dockerapp` folder, so we know that neither of them will be used.

Next, we can inspect the application directly from Docker Hub by running the following command (again, make sure to replace the Docker Hub ID with your own):

```
$ docker app inspect russmckendrick/mobycounterapp:0.1.0
```

We should see information on our application similar to that shown when we last ran the command locally. First up, let's create a new `docker-compose.yml` file using the Docker Hub-hosted version of the application. To do this, run the following command:

```
$ docker app render --output docker-compose.yml russmckendrick/
mobycounterapp:0.1.0
```

This will create a `docker-compose.yml` file in the current working folder; from there, we can then run the following:

```
$ docker-compose up -d
$ docker-compose ps
```

You will get a warning that the node is `running Swarm` mode; again, ignore this for now. You should see the two containers running, and your application will again be accessible at `http://localhost:8080`.

Next, let's launch the application again natively, but this time alongside our Docker Compose-launched version. To do this, run the following:

```
$ docker app install russmckendrick/mobycounterapp:0.1.0 --set
port=8181 --name mobycounterapp8181
```

As you can see, we are launching an application named `mobycounterapp8181`. We are also using the `--set` command to override the default port of `8080`, which we originally set in the `parameters.yml` file, and changing it to `8181`. If everything went as planned, you should be able to access the application at `http://localhost:8181`.

There is more functionality within Docker App. However, we are not quite ready to go into further details. We will return to Docker App in *Chapter 8, Docker Swarm* and *Chapter 11, Docker and Kubernetes*.

As mentioned at the top of this section, this feature is still in active development, and it is possible that the commands and functionality we have discussed so far may change in the future. But even at this development stage, I hope you can see the advantages of Docker App and how it is building on the solid foundations laid by Docker Compose.

Summary

I hope you have enjoyed this chapter on Docker Compose, and I hope that, like I did, you can see that it has evolved from being an incredibly useful third-party tool to an extremely important part of the core Docker experience.

Docker Compose introduces some key concepts as to how you should approach running and managing your containers. We will be taking these concepts one step further in *Chapter 8, Docker Swarm* and *Chapter 11, Docker and Kubernetes*, where we start to look at managing multiple Docker hosts and how we can distribute containers across them.

In the next chapter, we are going to move away from Linux-based containers and take a whistle-stop tour of Windows containers.

Questions

1. Docker Compose files use which open source format?
2. In our initial Moby counter Docker Compose file, which was the only flag that works exactly the same as its Docker **command-line interface** (**CLI**) counterpart?
3. True or false: You can only use images from the Docker Hub with your Docker Compose files.
4. By default, how does Docker Compose decide on the namespace to use?
5. Which flag do you add to docker-compose up to start the containers in the background?
6. What is the best way to run a syntax check on your Docker Compose files?
7. Explain the basic principle about how Docker App works.

Further reading

For details on Orchard Laboratories, see the following:

- Orchard Laboratories website: `https://web.archive.org/web/20171020135129/https://www.orchardup.com/`
- Orchard Laboratories joins Docker: `https://www.docker.com/blog/welcoming-the-orchard-and-fig-team/`

For a full Docker Compose compatibility matrix, see the following:

- Compose file versions and upgrading: `https://docs.docker.com/compose/compose-file/compose-versioning/`

For more information on the Docker App project, see the following:

- GitHub repository: `http://github.com/docker/app/`

Finally, here are some further links to a number of other topics that we have covered:

- YAML project home page: `http://www.yaml.org/`
- Docker sample repository: `https://github.com/dockersamples/`

6
Docker Machine, Vagrant, and Multipass

In this chapter, we will take a deeper look at Docker Machine. It can be used to easily launch and bootstrap Docker hosts targeting various platforms, including locally or in a cloud environment. We will also look at both Vagrant and Multipass, which are alternatives that can be used to launch local Docker hosts.

Let's take a look at what we will be covering in this chapter. We will be looking at the following topics:

- An introduction to Docker Machine
- Deploying local Docker hosts with Docker Machine
- Launching Docker hosts in the cloud using Docker Machine
- Using Vagrant and Multipass to launch local Docker hosts
- Introducing and using Multipass

Technical requirements

As in previous chapters, we will continue to use our local Docker installations. Again, the screenshots in this chapter will be from my preferred operating system, macOS. We will be looking at how we can use Docker Machine to launch Docker-based **virtual machines** (**VMs**) locally using VirtualBox, as well as in public clouds, so you will need an account with DigitalOcean if you would like to follow along with the examples in this chapter.

As before, the Docker commands we will be running will work on all three operating systems on which we have installed Docker so far. However, some of the supporting commands, which will be few and far between, may only apply to macOS- and Linux-based operating systems.

Check out the following video to see the Code in Action:
`https://bit.ly/2R6QQmd`

An introduction to Docker Machine

Before we roll our sleeves up and get stuck in with Docker Machine, we should take a moment to discuss what place it now occupies in the overall Docker ecosystem and what it actually is.

Docker Machine's biggest strength was that it provided a consistent interface to several public cloud providers, such as **Amazon Web Services** (**AWS**), **DigitalOcean**, **Microsoft Azure**, and **Google Cloud**, as well as self-hosted VM/cloud platforms (including **OpenStack** and **VMware vSphere**) to quickly launch and configure individual Docker hosts. Being able to target all of these technologies using a single command with minimal user interaction is a very big-time saver. If you need to quickly access a Docker host in AWS one day and then in DigitalOcean the next, you know you are going to get a consistent experience.

Also, it allows you to launch a local Docker host, which was useful for operating systems such as the non-Professional versions of Windows 10, where Docker could natively run because of a lack of hypervisor support.

As with Docker Compose, it used to be bundled with Docker for Windows and Docker for Mac— however, this has been dropped from recent releases. This is due to there being less of a requirement for end users to launch and manage individual Docker hosts. Now, people are using clustering technologies such as Docker Swarm or Kubernetes, or using a cloud provider's native Docker-based hosted tools, all of which we are going to be covering in detail across the remaining chapters.

> **Important note**
>
> A few of the tools we are going to be covering in this section of the chapter are now considered legacy, and support for them has started to be phased out. The reason they are mentioned is because for users with older hardware, they may the only way they can experience Docker.

We are going to start by quickly discussing one of these legacy tools.

Installing Docker Machine using Docker Toolbox

If you are running a version of macOS or Windows that does not support the versions of Docker for Windows and Docker for Mac we have been using in previous chapters, then you can download and install Docker Toolbox, which will install the Docker client, Docker Compose, and Docker Machine. You can download Docker Toolbox from `https://github.com/docker/toolbox/releases`; however, please do not install it if you are already running Docker, as it may create conflicts and create problems with your existing installation.

Docker Toolbox is now considered legacy, and the versions of Docker and the supporting tools that are installed are old; therefore, we will not be covering their installation here.

Installing Docker Machine using the command line

If you are already running Docker, before we can start using Docker Machine, we need to install it. Here is the command that you need to run to install Docker Machine on macOS and Linux, starting with macOS:

```
$ base=https://github.com/docker/machine/releases/download/
v0.16.2 &&
  curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/usr/
local/bin/docker-machine &&
  chmod +x /usr/local/bin/docker-machine
```

You would use a similar command for Linux, as follows:

```
$ curl -L https://github.com/docker/machine/releases/download/
v0.16.2/docker-machine-`uname -s`-`uname -m` >/tmp/docker-
machine &&
    chmod +x /tmp/docker-machine &&
    sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```

For Windows, the following command assumes that you have Git bash installed:

```
$ if [[ ! -d '$HOME/bin' ]]; then mkdir -p '$HOME/bin'; fi && \
curl -L https://github.com/docker/machine/releases/download/v0.16.2/docker-machine-Windows-x86_64.exe > '$HOME/bin/docker-machine.exe' && \
chmod +x '$HOME/bin/docker-machine.exe'
```

As you can see, all three commands are downloading an executable from the project's release page. To make sure that you are using the latest version, you can check `https://github.com/docker/machine/releases/`.

Now that we have Docker Machine installed on your chosen operating system, we can start to deploy VMs that run Docker.

Deploying local Docker hosts with Docker Machine

Before we journey out into the cloud experience, we are going to look at the basics of Docker Machine locally by launching it, using Oracle VirtualBox to provide the VM.

> **Important note**
>
> VirtualBox is a free virtualization product from Oracle. It allows you to install VMs across many different platforms and **central processing unit** (**CPU**) types. Download and install VirtualBox from `https://www.virtualbox.org/wiki/Downloads/`.

To launch the machine, all you need to do is run the following command:

```
$ docker-machine create --driver virtualbox docker-local
```

This will start the deployment, during which you will get a list of tasks that Docker Machine is running. To launch your Docker host, each host launched with Docker Machine goes through the same steps.

First of all, Docker Machine runs a few basic checks, such as confirming that VirtualBox is installed, and creating certificates and a directory structure in which it will store all of its files and VMs, as follows:

```
Creating CA: /Users/russ.mckendrick/.docker/machine/certs/ca.pem
Creating client certificate: /Users/russ.mckendrick/.docker/
machine/certs/cert.pem
Running pre-create checks...
(docker-local) Image cache directory does not exist, creating
it at /Users/russ.mckendrick/.docker/machine/cache...
```

It then checks for the presence of the image it will use for the VM. If it is not there, the image will be downloaded, as illustrated in the following code snippet:

```
(docker-local) No default Boot2Docker ISO found locally,
downloading the latest release...
(docker-local) Latest release for github.com/boot2docker/
boot2docker is v19.03.5
(docker-local) Downloading /Users/russ.mckendrick/.docker/
machine/cache/boot2docker.iso from https://github.com/
boot2docker/boot2docker/releases/download/v19.03.5/boot2docker.
iso...
(docker-local) 0%....10%....20%....30%....40%....50%....60%....
70%....80%....90%....100%
```

Once the checks have passed, it creates the VM using the selected driver, as illustrated in the following code snippet:

```
Creating machine...
(docker-local) Copying /Users/russ.mckendrick/.docker/machine/
cache/boot2docker.iso to /Users/russ.mckendrick/.docker/
machine/machines/docker-local/boot2docker.iso...
(docker-local) Creating VirtualBox VM...
(docker-local) Creating SSH key...
(docker-local) Starting the VM...
(docker-local) Check network to re-create if needed...
(docker-local) Found a new host-only adapter: 'vboxnet0'
(docker-local) Waiting for an IP...
Waiting for machine to be running, this may take a few
minutes...
```

As you can see, Docker Machine creates a unique **Secure Shell** (**SSH**) key for the VM. This means that you will be able to access the VM over SSH, but more on that later. Once the VM has booted, Docker Machine then makes a connection to the VM, as illustrated in the following code snippet:

```
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
```

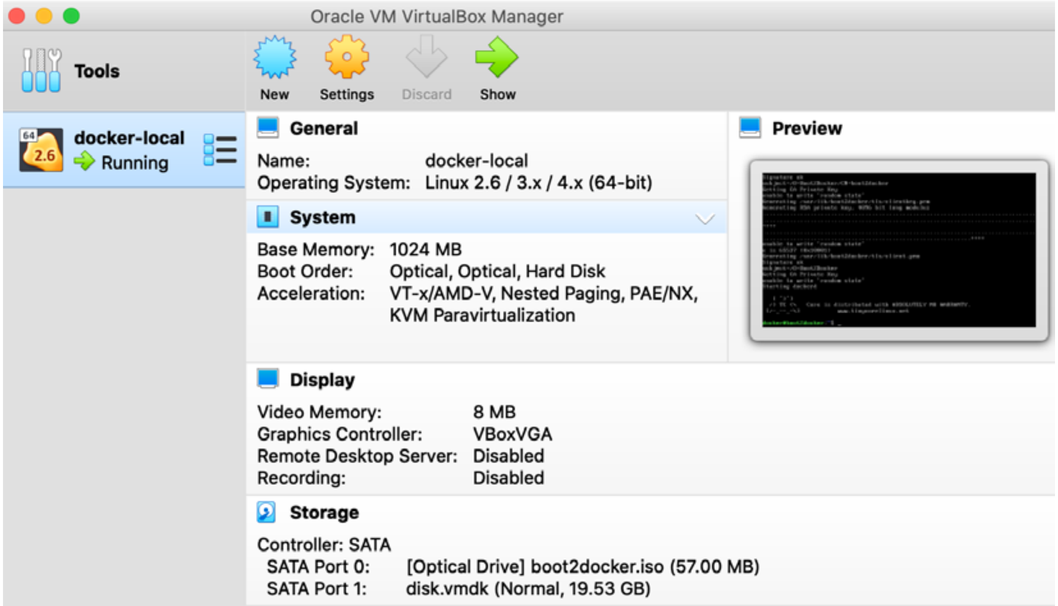
As you can see, Docker Machine detects the operating system being used and chooses the appropriate bootstrap script to deploy Docker. Once Docker is installed, Docker Machine generates and shares certificates between your localhost and the Docker host. It then configures the remote Docker installation for certificate authentication, meaning that your local client can connect to and interact with the remote Docker server.

Once Docker is installed, Docker Machine generates and shares certificates between your localhost and the Docker host. It then configures the remote Docker installation for certificate authentication, meaning that your local client can connect to and interact with the remote Docker server. This is illustrated in the following code snippet:

```
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine
running on this VM, run: docker-machine env docker-local
```

Finally, it checks whether your local Docker client can make the remote connection, and completes the task by giving you instructions on how to configure your local client to the newly launched Docker host.

If you open VirtualBox, you should be able to see your new VM, as illustrated in the following screenshot:



General

Name: docker-local
Operating System: Linux 2.6 / 3.x / 4.x (64-bit)

System

Base Memory: 1024 MB
Boot Order: Optical, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization

Display

Video Memory: 8 MB
Graphics Controller: VBoxVGA
Remote Desktop Server: Disabled
Recording: Disabled

Storage

Controller: SATA
SATA Port 0: [Optical Drive] boot2docker.iso (57.00 MB)
SATA Port 1: disk.vmdk (Normal, 19.53 GB)

Figure 6.1 – The Docker VM running in VirtualBox

Next, we need to configure our local Docker client to connect to the newly launched Docker host; as already mentioned in the output of launching the host, running the following command will show you how to make the connection:

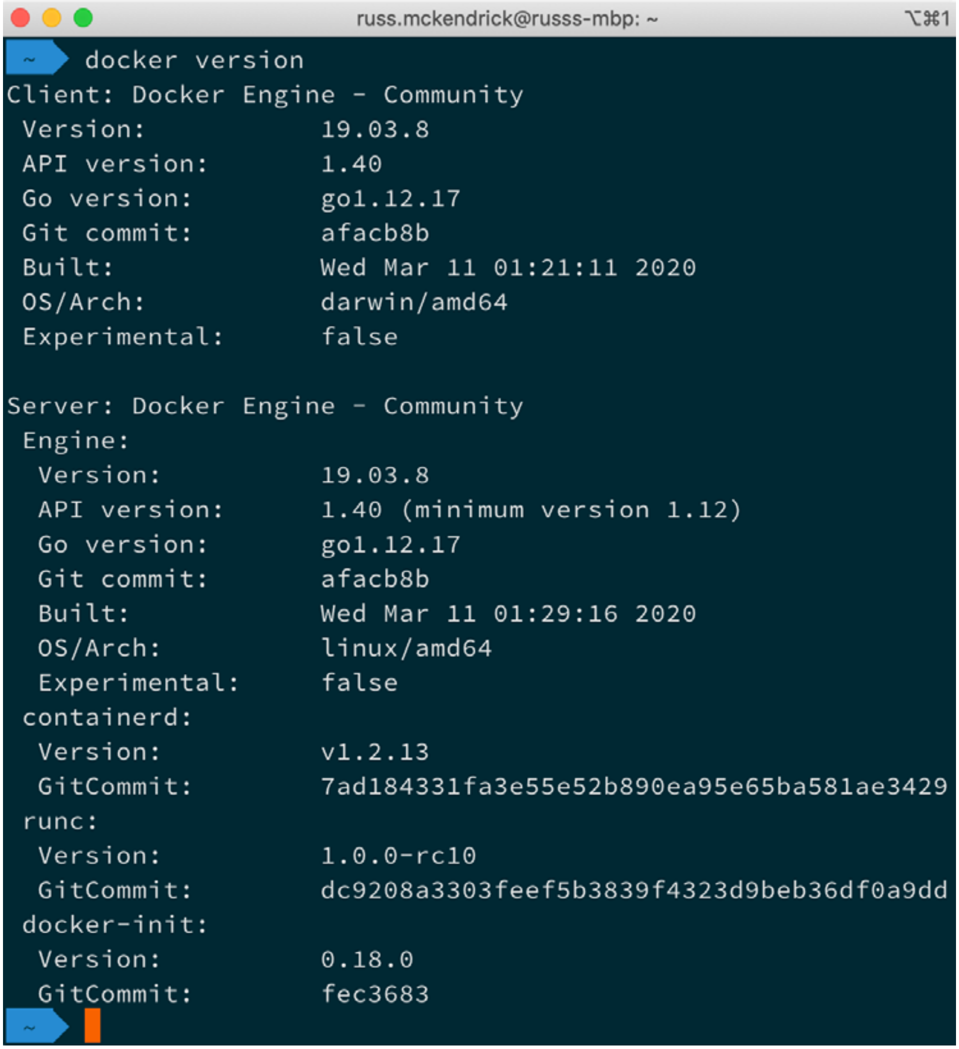
```
$ docker-machine env docker-local
```

This command returns the following:

```
export DOCKER_TLS_VERIFY='1'
export DOCKER_HOST='tcp://192.168.99.100:2376'
export DOCKER_CERT_PATH='/Users/russ.mckendrick/.docker/
machine/machines/docker-local'
export DOCKER_MACHINE_NAME='docker-local'
# Run this command to configure your shell:
# eval $(docker-machine env docker-local)
```

This overrides the local Docker installation by giving the IP address and port number of the newly launched Docker host, as well as the path to the certificates used for authentication. At the end of the output, it gives you a command to run and to configure your Terminal session in order to make the connection.

Before we run the command, let's run `docker version` to get information on the current setup, as follows:



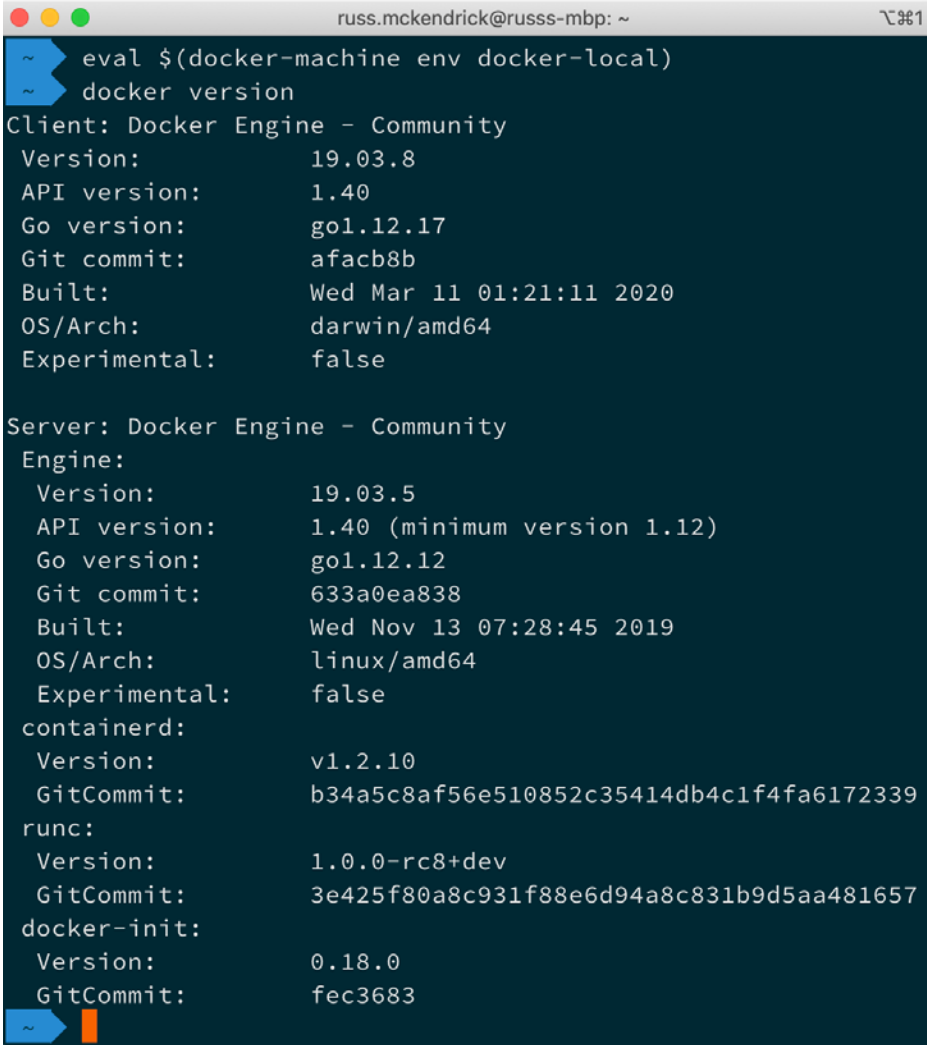
```
~  docker version
Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        afacb8b
 Built:             Wed Mar 11 01:21:11 2020
 OS/Arch:           darwin/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.8
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.17
  Git commit:       afacb8b
  Built:            Wed Mar 11 01:29:16 2020
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          v1.2.13
  GitCommit:        7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc:
  Version:          1.0.0-rc10
  GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
~ 
```

Figure 6.2 – Checking the versions on Docker on my local installation

This is basically the Docker for Mac installation I am running. Running the following command and then `docker version` again should show some changes to the server:

```
$ eval $(docker-machine env docker-local)
```

The output of the command is given in the following screenshot:



```
~  eval $(docker-machine env docker-local)
~  docker version
Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        afacb8b
 Built:             Wed Mar 11 01:21:11 2020
 OS/Arch:           darwin/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.5
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.12
  Git commit:       633a0ea838
  Built:            Wed Nov 13 07:28:45 2019
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          v1.2.10
  GitCommit:        b34a5c8af56e510852c35414db4c1f4fa6172339
 runc:
  Version:          1.0.0-rc8+dev
  GitCommit:        3e425f80a8c931f88e6d94a8c831b9d5aa481657
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
~
```

Figure 6.3 – Checking the versions of Docker on the new VM

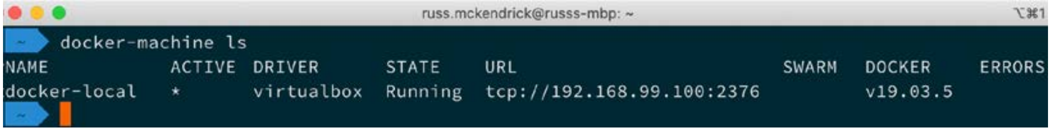
As you can see, the server launched by Docker Machine is pretty much in line with what we have installed locally; in fact, the only difference is that it is a few minor versions behind. As you can see, the Docker Engine binary on my Docker for Mac installation is running version `19.03.8` and the Docker Machine launched host is running `19.03.5`.

From here, we can interact with the Docker host in the same way as if it were a local Docker installation. Before we move on to launching Docker hosts in the cloud, there are a few other basic Docker Machine commands to cover.

The first command lists the currently configured Docker hosts, and is shown here:

```
$ docker-machine ls
```

The output of the command is given here:



```
docker-machine ls
NAME           ACTIVE  DRIVER      STATE    URL                         SWARM   DOCKER     ERRORS
docker-local   *       virtualbox  Running  tcp://192.168.99.100:2376           v19.03.5
```

Figure 6.4 – Listing the VMs

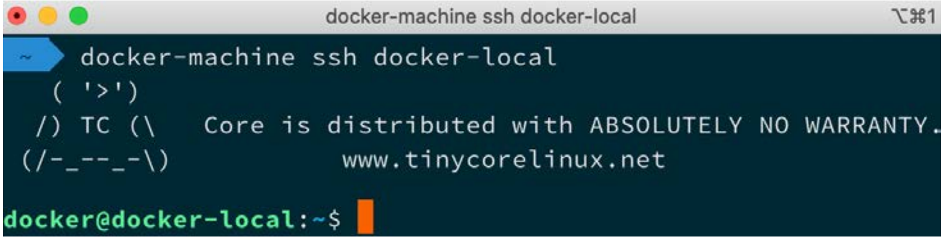
As you can see, it lists details of the machine name, the driver used, and the Docker endpoint **Uniform Resource Locator** (**URL**), as well as the version of Docker the hosts are running.

You will also notice that there is a `*` in the `ACTIVE` column; this indicates which Docker host your local client is currently configured to interact with. You can also find out the active machine by running `docker-machine active`.

The next command connects you to the Docker host using SSH, and is shown here:

```
$ docker-machine ssh docker-local
```

The output of the command is given here:



```
docker-machine ssh docker-local
   ( '>')
  /) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.
 (/-_--_-\)           www.tinycorelinux.net

docker@docker-local:~$
```

Figure 6.5 – Connecting to the VM using SSH

This is useful if you need to install additional software or configuration outside of Docker Machine. It is also useful if you need to look at logs, and so on. You can exit the remote shell by running `exit`. You can find out the IP address of your Docker host by running the following command once back on your local machine:

```
$ docker-machine ip docker-local
```

We will be using this command a lot throughout the chapter as part of other commands to get the IP addresses of our VMs. There are also commands for establishing more details about your Docker host, and these are shown in the following code snippet:

```
$ docker-machine inspect docker-local
$ docker-machine config docker-local
$ docker-machine status docker-local
$ docker-machine url docker-local
```

Finally, there are also commands to stop, start, restart, and remove your Docker host. Use the final command in the following code snippet to remove your locally launched host:

```
$ docker-machine stop docker-local
$ docker-machine start docker-local
$ docker-machine restart docker-local
$ docker-machine rm docker-local
```

Running the `docker-machine rm` command will prompt you to determine whether you really want to remove the instance, as illustrated here:

```
About to remove docker-local
WARNING: This action will delete both local reference and
remote instance.
Are you sure? (y/n): y
Successfully removed docker-local
```

Now that we have had a very quick rundown of the basics, let's try something more adventurous.

Launching Docker hosts in the cloud using Docker Machine

In this section, we are going to take a look at just one of the public cloud drivers supported by Docker Machine. As already mentioned, there are plenty available, but part of the appeal of Docker Machine is that it offers consistent experiences, so there are not too many differences between the drivers.

We are going to be launching a Docker host in **DigitalOcean** using Docker Machine. To do this, we need an **application programming interface** (**API**) access token with the necessary permissions to access and launch resources within your DigitalOcean account. Rather than explaining how to generate one here, you can follow the instructions at `https://www.digitalocean.com/help/api/`.

> **Important note**
>
> Launching a Docker host using the API token will incur a cost; ensure you keep track of the Docker hosts you launch. Details on DigitalOcean's pricing can be found at `https://www.digitalocean.com/pricing/`. Also, keep your API token secret as it could be used to gain unauthorized access to your account. All of the tokens used in this chapter have been revoked.

The first thing we are going to do is set our token as an environment variable so that we don't have to keep using it. To do this, run the following command:

```
$
DOTOKEN=191e004d9a58b964198ab1e8253fc2de367a70fceb9847b7fd44ebf
```

Make sure you replace the API token with your own.

> **Important note**
>
> Due to the additional flags that we need to pass to the `docker-machine` command, I will be using / to split the command across multiple lines to make it more readable.

The command to launch the Docker host in DigitalOcean is as follows:

```
$ docker-machine create \
      --driver digitalocean \
      --digitalocean-access-token $DOTOKEN \
      docker-digitalocean
```

As the Docker host is a remote machine, it will take a little while to launch, configure, and be accessible. As you can see from the following output, there are also a few changes to how Docker Machine bootstraps the Docker host:

```
Running pre-create checks...
Creating machine...
(docker-digitalocean) Creating SSH key...
(docker-digitalocean) Creating Digital Ocean droplet...
(docker-digitalocean) Waiting for IP address to be assigned to
the Droplet...
Waiting for machine to be running, this may take a few
minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(systemd)...
Installing Docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine
running on this VM, run: docker-machine env docker-digitalocean
```

Once launched, you should be able to see the Docker host in your DigitalOcean control panel, as illustrated in the following screenshot:

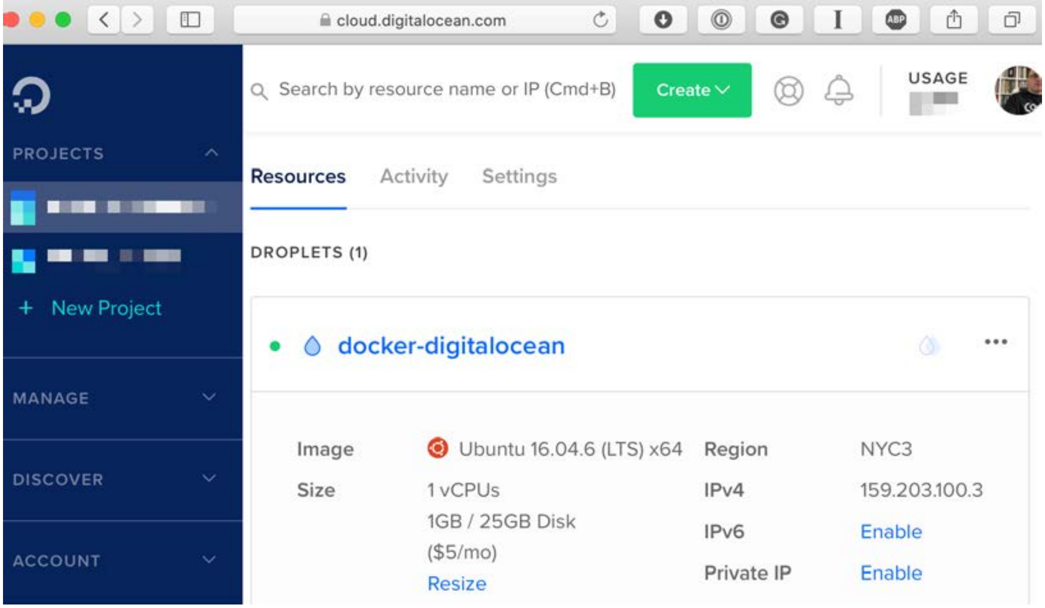


Image	Ubuntu 16.04.6 (LTS) x64	Region	NYC3
Size	1 vCPUs 1GB / 25GB Disk ($5/mo)	IPv4	159.203.100.3
		IPv6	Enable
		Private IP	Enable

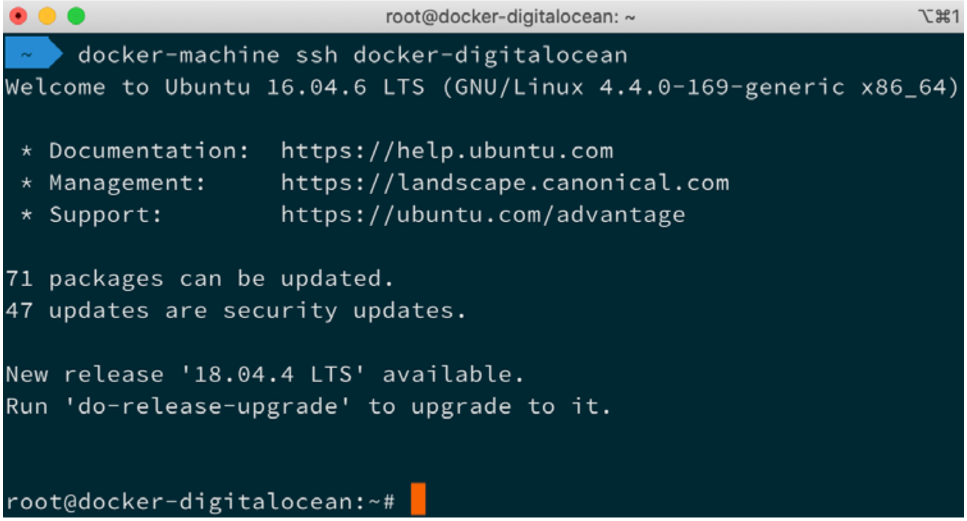
Figure 6.6 – Viewing the droplet in the DigitalOcean portal

Reconfigure your local client to connect to the remote host by running the following command:

```
$ eval $(docker-machine env docker-digitalocean)
```

Also, you can run `docker version` and `docker-machine inspect docker-digitalocean` to find out more information about the Docker host.

Finally, running `docker-machine ssh docker-digitalocean` will SSH you into the host. As you can see from the following output, and also from the output when you first launched the Docker host, there is a difference in the operating system used:



```
~ docker-machine ssh docker-digitalocean
Welcome to Ubuntu 16.04.6 LTS (GNU/Linux 4.4.0-169-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

71 packages can be updated.
47 updates are security updates.

New release '18.04.4 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


root@docker-digitalocean:~#
```

Figure 6.7 – SSHing into our DigitalOcean machine

You can exit the remote shell by running `exit`. As you can see, we didn't have to tell Docker Machine which operating system to use, the size of the Docker host, or even where to launch it. That is because each driver has some pretty sound defaults.

Adding these defaults to our command makes it look like the following:

```
docker-machine create \
     --driver digitalocean \
     --digitalocean-access-token $DOTOKEN \
     --digitalocean-image ubuntu-16-04-x64 \
     --digitalocean-region nyc3 \
     --digitalocean-size 512mb \
     --digitalocean-ipv6 \
     --digitalocean-backups \
     --digitalocean-ssh-user root \
     --digitalocean-ssh-port 22 \
     docker-digitalocean
```

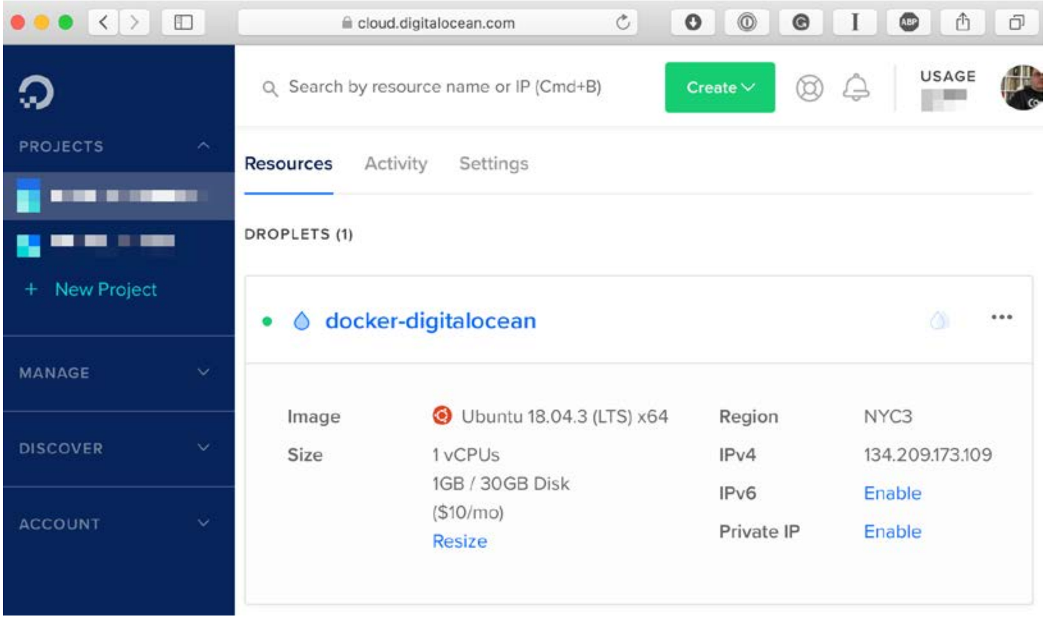
As you can see, there is scope for you to customize the size, region, and operating system, and even the network your Docker host is launched with.

> **Important note**
>
> The command in the code snippet that follows will give an error if you have a droplet already launched.

Let's say we wanted to change the operating system and the size of the droplet. In this instance, we can run the following:

```
$ docker-machine create \
      --driver digitalocean \
      --digitalocean-access-token $DOTOKEN \
      --digitalocean-image ubuntu-18-04-x64 \
      --digitalocean-size 1gb \
      docker-digitalocean
```

As you can see in the **DigitalOcean** control panel, this launches a machine that looks like the following:



DROPLETS (1)

docker-digitalocean

Image: Ubuntu 18.04.3 (LTS) x64

Size: 1 vCPUs, 1GB / 30GB Disk ($10/mo) Resize

Region: NYC3

IPv4: 134.209.173.109

IPv6: Enable

Private IP: Enable

Figure 6.8 – Viewing the droplet with a different specification

You can remove the DigitalOcean Docker host by running the following command:

```
$ docker-machine rm docker-digitalocean
```

It will take a minute or two to properly remove the host.

> **Important note**
>
> Please double-check in the DigitalOcean portal that your host has been properly terminated; otherwise, you may incur unexpected costs.

That concludes our look at Docker Machine. Before we move on, let's discuss how secure it is.

Docker Machine summary

As already mentioned, Docker Machine is now considered a legacy tool. As well as the aforementioned reason that people no longer want to launch single Docker hosts, there are also some other reasons for the change in direction.

The most important one is security. As you may have noticed, when Docker Machine was launching our Docker hosts it generated not only an SSH key but also certificates, and then configured the Docker server component with these, meaning that our local Docker client could interact with the host. Now, with the localhost this was not a problem; however, on the DigitalOcean-hosted instance, we were exposing our Docker server to the internet.

While it was securely configured, if there were ever to be a zero-day exploit with the version of Docker that was installed we could end up with a few problems, as untrusted third parties may be able to take over our Docker installation. Because of this, I would recommend only using Docker Machine to quickly spin up an instance, test something, and then spin the instance down.

We will be looking at some better options for running containerized applications on clouds in *Chapter 10*, *Running Docker in Public Clouds,* and *Chapter 13*, *Running Kubernetes in Public Clouds*.

Let's now look at a few alternatives to using Docker Machine to launch a local Docker host.

Using Vagrant and Multipass to launch local Docker hosts

Before we finish the chapter, we are going to look at two different tools that can be used to launch a machine locally, and then provision and configure a local Docker host for you to experiment with in later chapters.

Introducing and using Vagrant

Vagrant, by **HashiCorp**, is the granddaddy of host management tools. First released in early 2010, what started as a side project for **Mitchell Hashimoto** has turned into a quite important piece of software in modern software development as it allows developers to quickly and easily launch and remove local VMs for use as development environments. In May 2014, version 1.6 was released, which introduced support for Docker.

The installation differs slightly depending on your operating system. If you are running macOS, then you can use Homebrew and Cask. The command to use these is shown in the following code snippet:

```
$ brew cask install vagrant
```

Windows and Linux users can download the installers from `https://www.vagrantup.com/downloads.html`. There, you can find both 32-bit and 64-bit `MSI`, `deb`, and `RPM` packages for your preferred operating system, as well as 64-bit static binaries for macOS and Linux if you would prefer not to use a package manager.

Vagrant uses a `Vagrantfile`, which defines what the VM looks like; the machines themselves, packaged up into a format called a Vagrant Box—a box containing the machine image; and also, the metadata needed to launch the host using VirtualBox, VMWare, or any other supported hypervisor or cloud service. Boxes are typically downloaded from Vagrant Cloud and are created by another HashiCorp product called **Packer**.

To create a `Vagrantfile`, you can simply run the following command in a folder in which you wish to store the files relating to your Vagrant Box:

```
$ vagrant init ubuntu/bionic64
```

The default `Vagrantfile` is heavily commented; however, the majority of the options are commented out. Removing these gives us something that should look like the following:

```
Vagrant.configure('2') do |config|
  config.vm.box = 'ubuntu/bionic64'
end
```

As you can see, there isn't much to it and there certainly isn't anything to do with Docker, so let's fix that. Update the `Vagrantfile` so that it looks like the following:

```
Vagrant.configure('2') do |config|
  config.vm.box = 'ubuntu/bionic64'
  config.vm.provision :docker
end
```

Once updated, run the following command and wait:

```
$ vagrant up
```

If you don't already have the `ubuntu/bionic64` box downloaded, then Vagrant will download that for you. Once downloaded, it will launch a VM in VirtualBox, which is its default provider. You should see something like the following as an output:

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/bionic64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/bionic64' version
'20200402.0.0' is up to date...
==> default: Setting the name of the VM: vagrant_
default_1586094728360_48806
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on
configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
```

```
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
```

Once the VM has booted, Vagrant will SSH into the machine and replace the default Vagrant key file with a more secure self-generated one, as illustrated in the following code snippet:

```
==> default: Waiting for machine to boot. This may take a few
minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default:
    default: Vagrant insecure key detected. Vagrant will
automatically replace
    default: this with a newly generated keypair for better
security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's
present...
    default: Key inserted! Disconnecting and reconnecting using
new SSH key...
==> default: Machine booted and ready!
```

Now that the SSH key is secure, Vagrant will check that the VM is running the VirtualBox guest additions. These will allow it to share your local machine's filesystem with the VM. The process is illustrated in the following code snippet:

```
==> default: Checking for guest additions in VM...
    default: The guest additions on this VM do not match the
installed version of
    default: VirtualBox! In most cases this is fine, but in
rare cases it can
    default: prevent things such as shared folders from working
properly. If you see
    default: shared folder errors, please make sure the guest
additions within the
    default: VM match the version of VirtualBox you have
```

```
installed on
    default: your host and reload your VM.
    default:
    default: Guest Additions Version: 5.2.34
    default: VirtualBox Version: 6.1
==> default: Mounting shared folders...
    default: /vagrant => /Users/russ.mckendrick/vagrant
```

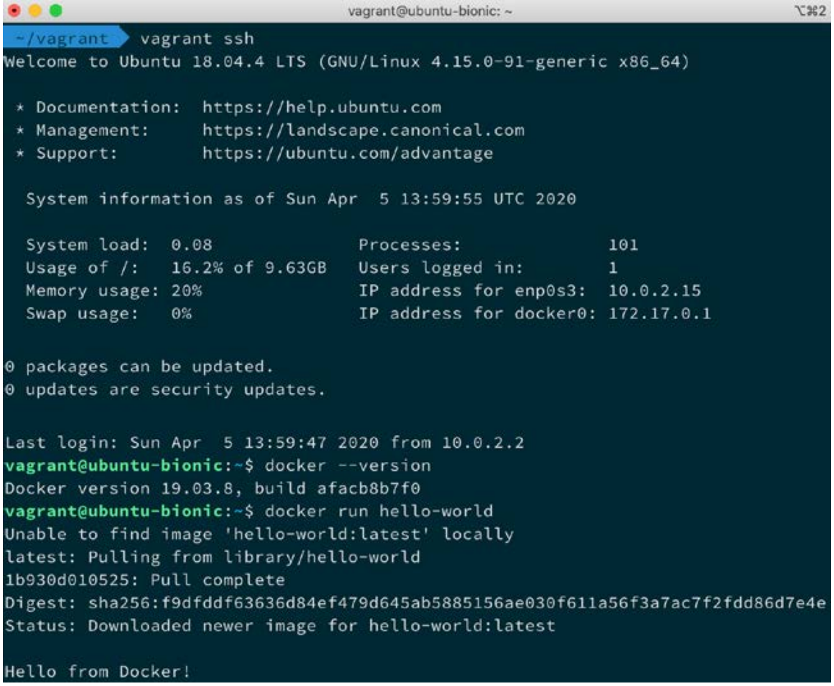
The final step is to run the provisioner we added to our `Vagrantfile`, as follows:

```
==> default: Running provisioner: docker...
    default: Installing Docker onto machine...
```

So, we now have an Ubuntu 18.04 (Bionic Beaver) VM with Docker installed. To access the VM, simply run the following command:

```
$ vagrant ssh
```

This will open an SSH session to the VM. From there, you can use Docker, as we have been doing when running it locally. See the following terminal output for an example:



```
 ~/vagrant  vagrant ssh
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-91-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sun Apr  5 13:59:55 UTC 2020

  System load:  0.08              Processes:              101
  Usage of /:   16.2% of 9.63GB   Users logged in:        1
  Memory usage: 20%               IP address for enp0s3:  10.0.2.15
  Swap usage:   0%                IP address for docker0: 172.17.0.1

0 packages can be updated.
0 updates are security updates.


Last login: Sun Apr  5 13:59:47 2020 from 10.0.2.2
vagrant@ubuntu-bionic:~$ docker --version
Docker version 19.03.8, build afacb8b7f0
vagrant@ubuntu-bionic:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f611a56f3a7ac7f2fdd86d7e4e
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

Figure 6.9 – Logging in to our Vagrant Box and interacting with Docker

Once you have finished with the VM, type `exit` to return to your host machine's Command Prompt, and then run the following command to terminate the VM:

```
$ vagrant destroy
```

This will remove all of the resources associated with the VM. You will see the following messages:

```
    default: Are you sure you want to destroy the 'default' VM? [y/N] y
==> default: Forcing shutdown of VM...
==> default: Destroying VM and associated drives...
```

Should you wish to just stop the VM, you can use the following commands to stop and start it:

```
$ vagrant stop
$ vagrant up
```

Now that we have learned how to create a VM and provision Docker using Vagrant, let's look at an alternative called **Multipass**.

Introducing and using Multipass

Multipass is a tool provided by *Canonical*, the makers of Ubuntu, to quickly launch and manage multiple Ubuntu VMs locally. It works slightly differently from Vagrant in that rather than defaulting to using VirtualBox, it will use your operating system's default hypervisor. Where one is not available, as with non-Professional versions of Windows 10, it has the ability to fall back to using VirtualBox—more on that in a moment.

To install Multipass on macOS, you can run the following command, which again uses Homebrew and Cask:

```
$ brew cask install multipass
```

To install on an existing Ubuntu desktop, you can use the Snap package manager by running the following command:

```
$ snap install multipass --classic
```

Finally, to install on Windows, you can download the installer from the project's release page at GitHub, where you will also find the macOS installer, at `https://github.com/canonical/multipass/releases`.

Once installed, if you need to use VirtualBox over your local machine's native hypervisor, you can run the following command:

```
$ multipass set local.driver=virtualbox
```

Now that we have Multipass and installed, and—if needed—configured to use VirtualBox, we can start a VM up and install Docker. To launch a VM, simply run the following command:

```
$ multipass launch --name docker-host
```

Once launched, we can run the following command to bootstrap Docker onto the newly created VM:

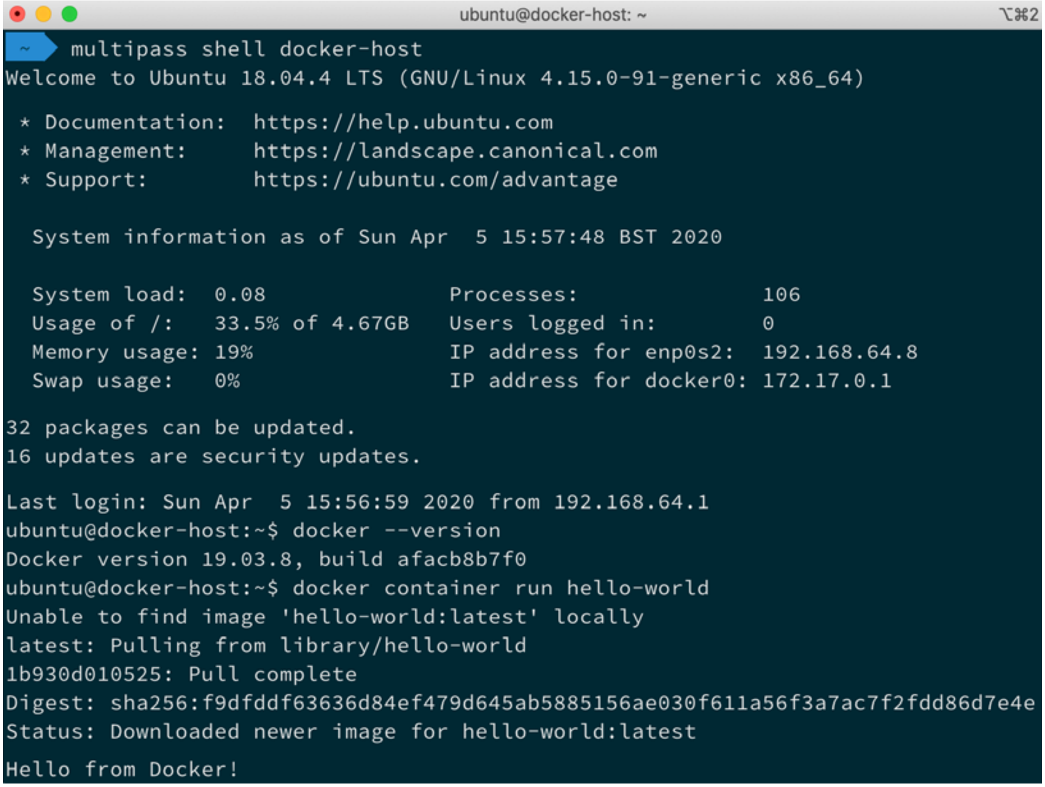
```
$ multipass exec docker-host -- /bin/bash -c 'curl -s https://
get.docker.com | sh - && sudo usermod -aG docker ubuntu'
```

This will download and install the latest community version of Docker. Once installed, it will then add the `ubuntu` user to the `docker` group, meaning that when connected to the VM we will be ready to use Docker straight away. You will see the output of the commands being executed during the installation, and, once complete, you will be returned to your local machine's shell.

To connect to the VM, run the following command:

```
$ multipass shell docker-host
```

Once connected, you will notice that by default, and as with our Vagrant VM, Multipass has downloaded and launched an `Ubuntu 18.04` (Bionic Beaver) VM. Once logged in, you will be able to run Docker commands as expected. The process is illustrated in the following screenshot:



```
~  multipass shell docker-host
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-91-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sun Apr  5 15:57:48 BST 2020

  System load:  0.08              Processes:              106
  Usage of /:   33.5% of 4.67GB   Users logged in:        0
  Memory usage: 19%               IP address for enp0s2:  192.168.64.8
  Swap usage:   0%                IP address for docker0: 172.17.0.1

32 packages can be updated.
16 updates are security updates.

Last login: Sun Apr  5 15:56:59 2020 from 192.168.64.1
ubuntu@docker-host:~$ docker --version
Docker version 19.03.8, build afacb8b7f0
ubuntu@docker-host:~$ docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f611a56f3a7ac7f2fdd86d7e4e
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

Figure 6.10 – SSHing to our Multipass VM and interacting with Docker

As with Vagrant, once you have finished, type in `exit`, and you will be returned to your local machine's shell. To remove the `docker-host` VM and all of its resources, you can run the following command:

```
$ multipass delete docker-host --purge
```

If you want to stop and start the VM, you can use the following:

```
$ multipass stop docker-host
$ multipass start docker-host
```

As you can see, using Multipass is a nice and simple way to both launch and interact with Ubuntu VMs locally on any host.

Summary

In this chapter, we looked at how to use Docker Machine to create Docker hosts locally on VirtualBox and reviewed the commands you can use to both interact with and manage your Docker Machine-launched Docker hosts. We then looked at how to use Docker Machine to deploy Docker hosts to a cloud environment—namely, **DigitalOcean**.

We also discussed why using Docker Machine may also not be a great idea, depending on the age of your host machine. Because of this, we also discussed how to launch and configure a local Docker host using both **Vagrant** and **Multipass**.

In later chapters, we are going to move away from interacting with a single Docker host and move on to launching and running multiple Docker hosts. However, before we do so, we are first going to move away from Linux-based containers and take a whistle-stop tour of Windows containers in the next chapter. Don't worry if you don't have a Windows-based host, as we will be using Vagrant to launch one.

Questions

1. Which flag, when running `docker-machine create`, lets you define which service or provider Docker Machine uses to launch your Docker host?
2. True or false: Running `docker-machine env my-host` will reconfigure your local Docker client to interact with `my-host`.
3. True or false: Multipass has native support for Docker out of the box.
4. Name the HashiCorp tool that can be used to create Vagrant Boxes.
5. Explain why using Docker Machine to create Docker hosts in the cloud is no longer considered best practice.

Further reading

For information on the various tools we have used and referred to in this chapter, refer to the following:

- Homebrew: `https://brew.sh`
- Cask: `https://github.com/Homebrew/homebrew-cask`
- Git BASH for Windows: `https://gitforwindows.org`
- VirtualBox: `https://www.virtualbox.org`
- Docker Machine: `https://docs.docker.com/machine/`
- Docker Toolbox: `https://docs.docker.com/toolbox/overview/`
- DigitalOcean: `https://www.digitalocean.com/`
- Vagrant: `https://www.vagrantup.com`
- Vagrant Cloud: `https://app.vagrantup.com/boxes/search`
- Packer: `https://packer.io`
- HashiCorp: `https://www.hashicorp.com`
- Multipass: `https://multipass.run`
- Snap: `https://snapcraft.io`
- Canonical: `https://canonical.com`

Section 2: Clusters and Clouds

In this section, we will be taking everything we have learned in the first part and applying it to the two main Docker clustering technologies, Docker Swarm and Kubernetes, as well as looking at running containers on public clouds.

This section comprises the following chapters:

Chapter 7, *Moving from Linux to Windows Containers*

Chapter 8, *Clustering with Docker Swarm*

Chapter 9, *Portainer – A GUI for Docker*

Chapter 10, Running Docker in Public Clouds

Chapter 11, Clustering with Docker and Kubernetes

Chapter 12, Discovering Other Kubernetes Options

Chapter 13, Running Kubernetes in Public Clouds

7
Moving from Linux to Windows Containers

In this chapter, we will discuss and take a look at Windows containers. Microsoft has embraced containers as a way of deploying older applications on new hardware. Unlike Linux containers, Windows containers are only available on Windows-based Docker hosts.

We will be covering the following topics:

- An introduction to Windows containers
- Setting up your Docker host for Windows containers
- Running Windows containers
- A Windows container Dockerfile
- Windows containers and Docker Compose

Technical requirements

In this chapter, the containers we will be launching will only work on a Windows Docker host. We will be using VirtualBox and Vagrant on macOS- and Linux-based machines to assist in getting a Windows Docker host up and running.

Check out the following video to see the Code in Action:
`https://bit.ly/2DEwopT`

An introduction to Windows containers

As someone who has been using mostly macOS and Linux computers and laptops alongside Linux servers pretty much daily for the past 20 years, coupled with the fact that my only experience of running Microsoft Windows was the Windows XP and Windows 10 gaming PCs I have had (along with the odd Windows server I was unable to avoid at work), the advent of Windows containers was an interesting development.

Now, I would never have classed myself as a Linux/Unix fanboy; however, Microsoft's actions over the last few years have surprised even me. Back in 2014, at one of its Azure events, Microsoft declared "Microsoft Linux", and it hasn't looked back since.

Some notable headlines since Microsoft's declaration of love for Linux include the following:

- Linux is a first-class citizen in Microsoft Azure.
- .NET Core is cross-platform, meaning that you can run your .NET applications on Linux, macOS, and Windows.
- SQL Server has been available on Linux for a few years.
- You can run Linux shells, such as Ubuntu, on Windows 10 Professional machines.
- PowerShell has been ported to Linux and macOS.
- It has developed cross-platform tools, such as Visual Studio Code, and open sourced them.
- It acquired GitHub for $7.5 billion!!

It is clear that the Microsoft of old, where former **chief executive officer** (**CEO**) Steve Ballmer famously roasted both the open source and Linux communities by calling them something that would not be appropriate to repeat here, has gone.

Hence, the announcement made in October 2014, months after Microsoft publicly declared its love for Linux, that Docker and Microsoft were forming a partnership to drive the adoption of containers on Windows-based operating systems such as Windows 10 Professional and Windows Server 2016 came as no surprise to anyone.

So, what are Windows containers?

Well, on the face of it, they are no different from Linux containers. The work by Microsoft on the Windows kernel has introduced the same process isolation as found on Linux. Also, like Linux containers, this isolation extends to a sandboxed filesystem and even a Windows registry.

As each container is effectively a fresh Windows Core or Windows Nano installation, which, in turn, are cut-down Windows Server images (think Alpine Linux but for Windows), installation administrators can run multiple dockerized applications on the same host without having to worry about any custom registry changes or requirements clashing and causing problems.

Couple this with the same ease of use supplied by the Docker command-line client, and administrators have a way to migrate their legacy applications to more modern hardware, and also to host operating systems without the worries and overhead of having to manage multiple **virtual machines** (**VMs**) running older unsupported versions of Windows.

There is also another layer of isolation provided by Windows containers. **Hyper-V isolation** runs the container processes within a minimal hypervisor when the container is started. This further isolates the container processes from the host machine. However, there is a small cost of additional resources that are needed for each container running with Hyper-V isolation, while these containers will also have an increased start time as the hypervisor needs to be launched before the container can be started.

While Hyper-V isolation does use Microsoft's hypervisor, which can be found in both Windows Server and Desktop editions, as well as the Xbox One system software, you can't manage Hyper-V isolated containers using the standard Hyper-V management tools. You have to use Docker.

After all the work and effort Microsoft had to put into enabling containers in the Windows kernel, why did they choose Docker over just creating their own container management tool?

Docker had already established itself as the go-to tool for managing containers with a set of proven **application programming interfaces** (**APIs**) and a large community. Also, it was an open source application, which meant that Microsoft could not only adapt it for use on Windows but also contribute to its development.

The following diagram gives an overview of how Docker on Windows works:

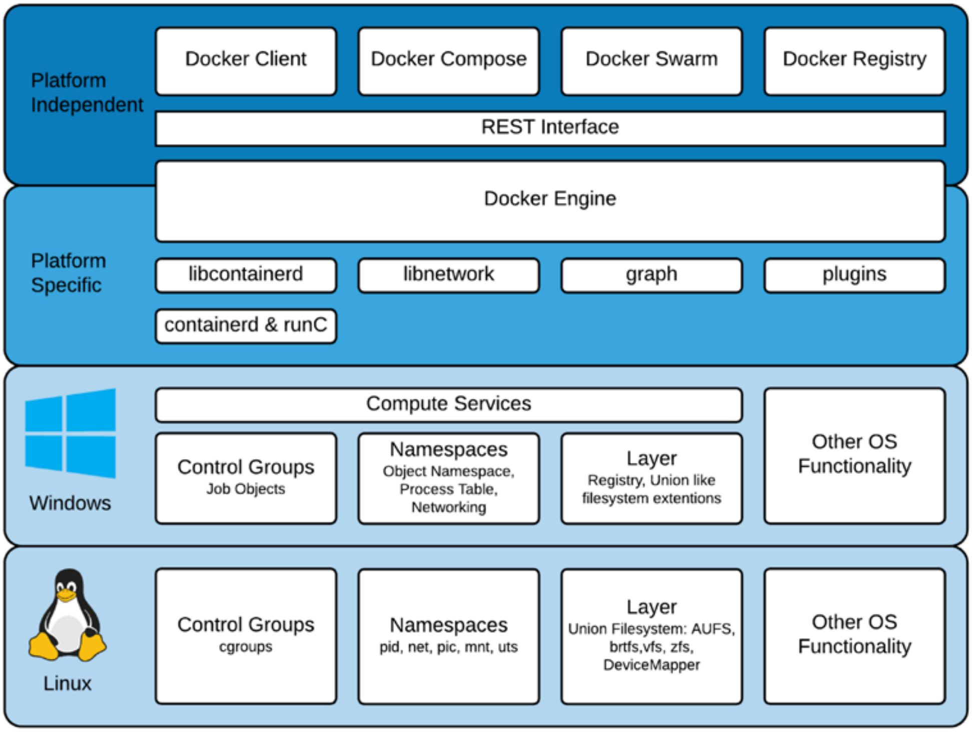


Figure 7.1 – Docker on Windows overview

Notice that I said Docker *on* Windows, not Docker *for* Windows; they are very different products. Docker on Windows is the native version of the Docker engine and client that interacts with the Windows kernel to provide Windows containers. Docker for Windows is a native-as-possible experience for developers to run both Linux and Windows containers on their desktops.

Now, let's look at preparing your host so that we can run Windows containers.

Setting up your Docker host for Windows containers

As you may have guessed, you are going to need access to a Windows host running Docker. Don't worry too much if you are not running a Windows 10 Professional machine; there are ways in which you can achieve this on macOS and Linux. Before we talk about those, let's look at how you can run Windows containers on Windows 10 Professional with your Docker for Windows installation.

Enabling Windows Container Support on Windows 10 Professional

Windows 10 Professional supports Windows containers out of the box. By default, however, it is configured to run Linux containers. To switch from running Linux containers to Windows containers, right-click on the **Docker** icon in your system tray and select **Switch to Windows containers…** from the menu, as illustrated in the following screenshot:

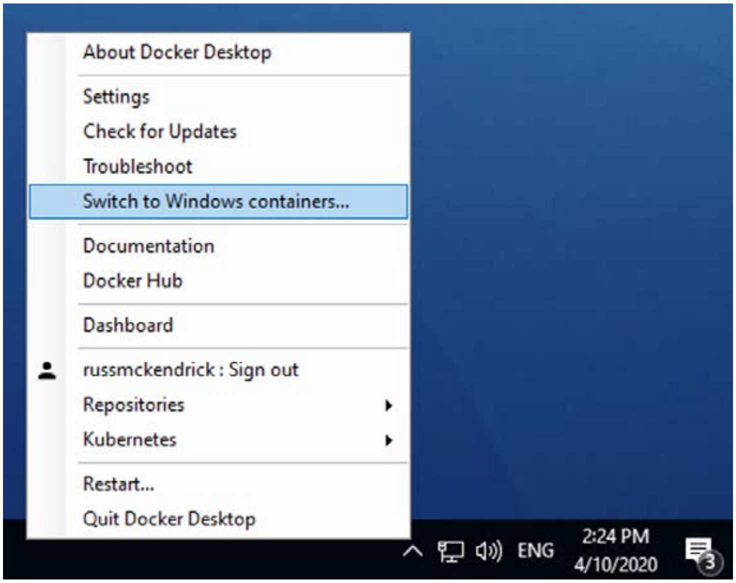
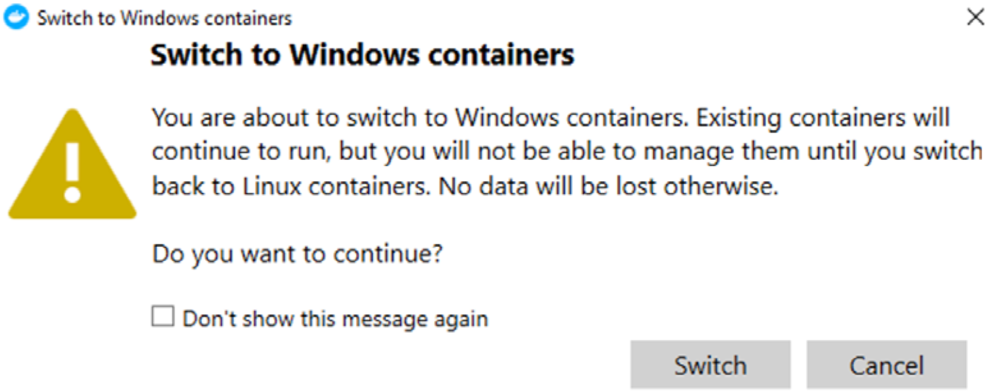


Figure 7.2 – Switching to Windows containers

This will bring up the following prompt:



Switch to Windows containers

You are about to switch to Windows containers. Existing containers will continue to run, but you will not be able to manage them until you switch back to Linux containers. No data will be lost otherwise.

Do you want to continue?

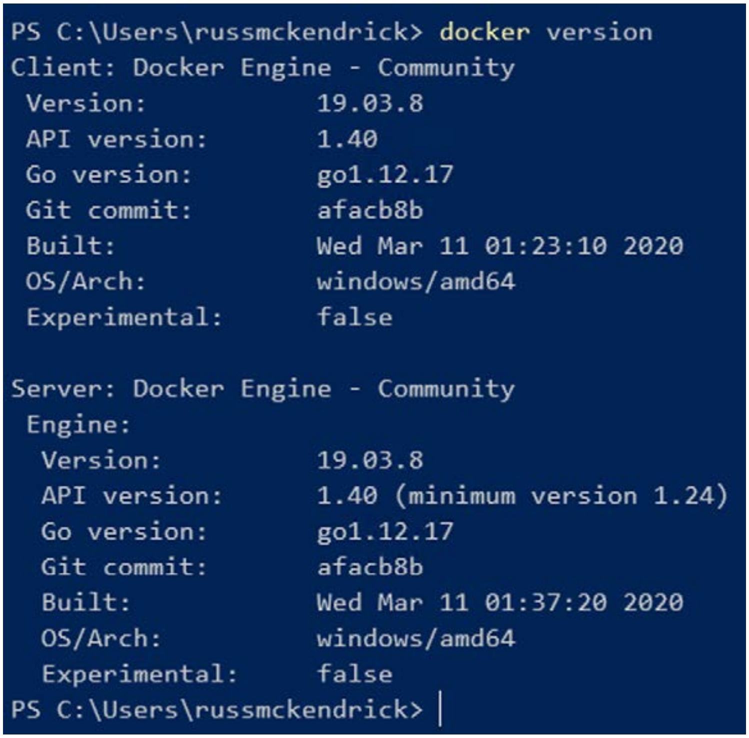
□ Don't show this message again

Figure 7.3 – An important note on what happens to your Linux containers

Hit the **Switch** button and, after a few seconds, you will now be managing Windows containers. You can see this by opening up Command Prompt on your host and running the following command:

```
$ docker version
```

This can be seen from the following output:



```
PS C:\Users\russmckendrick> docker version
Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        afacb8b
 Built:             Wed Mar 11 01:23:10 2020
 OS/Arch:           windows/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.8
  API version:      1.40 (minimum version 1.24)
  Go version:       go1.12.17
  Git commit:       afacb8b
  Built:            Wed Mar 11 01:37:20 2020
  OS/Arch:          windows/amd64
  Experimental:     false
PS C:\Users\russmckendrick> |
```

Figure 7.4 – Checking the output of running docker version

The Docker Engine has an `OS/Arch of windows/amd64` version, rather than the `linux/amd64` version we have been used to seeing up until now. So, that covers Windows 10 Professional. But what about people like me who prefer macOS or Linux?

Up and running on MacOS and Linux

To get access to Windows containers on macOS and Linux machines, we will be using the excellent resources put together by Stefan Scherer. In the `chapter07` folder of the repository that accompanies this book, there is a forked version of Stefan's `docker-windows-box` repo as a Git submodule, which contains all of the files you need to get up and running with Windows containers on macOS.

To check out the forked version, you will need to run the following command within the repository folder:

```
$ git submodule update --init --recursive
```

Finally, before we start to launch the VM, you will need the following tools: Vagrant by HashiCorp, and VirtualBox by Oracle, which we covered in the last chapter.

We do, however, need to install a `vagrant` plugin. To do this, run the following command:

```
$ vagrant plugin install vagrant-reload
```

Once the plugin is installed, we can start using Docker on Windows by opening a Terminal, going to the `chapter07/docker-machine` repository folder, and running the following command:

```
$ vagrant up
```

This will download a VirtualBox Windows Server 2019 `Core Eval` image that contains everything needed to get you up and running with Windows containers. The download is just over 6 **gigabytes** (**GB**), so please make sure that you have the bandwidth and disk space needed to run the image.

Vagrant will launch the image and configure Docker on the VM, along with a few other sensible defaults such as the Atom **integrated development environment** (**IDE**), Docker Compose, Docker Machine, and Git. Once the VM has launched, open your preferred Microsoft **Remote Desktop Protocol** (**RDP**) client, and then run the following command:

```
$ vagrant rdp
```

If you are prompted for the password, enter `vagrant`, and you will be logged in to your newly launched Windows 2019 Server environment, with all of the tools you need to run Docker in Windows installed and ready to go.

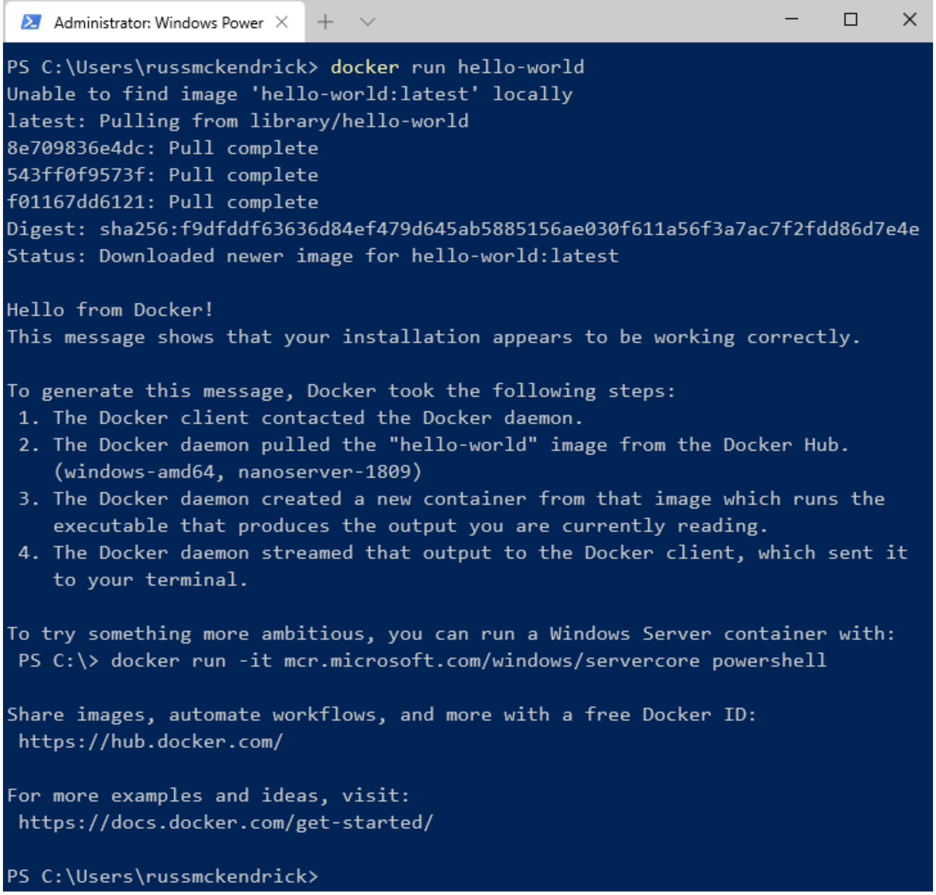
Also, if you don't want to run something locally, a Windows 10 Professional instance in Azure has all of the necessary components enabled to be able to run Docker for Windows, which—as discussed in the previous section—allows you to run Windows containers, which we will be looking at next.

Running Windows containers

As already hinted at in the first part of this chapter, launching and interacting with Windows containers using the Docker command-line client is no different from what we have been running so far. Let's test this by running the `hello-world` container, as follows:

```
$ docker container run hello-world
```

Just as before, this will download the `hello-world` container and return a message, as illustrated in the following screenshot:



```
PS C:\Users\russmckendrick> docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
8e709836e4dc: Pull complete
543ff0f9573f: Pull complete
f01167dd6121: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f611a56f3a7ac7f2fdd86d7e4e
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (windows-amd64, nanoserver-1809)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run a Windows Server container with:
 PS C:\> docker run -it mcr.microsoft.com/windows/servercore powershell

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

PS C:\Users\russmckendrick>
```

Figure 7.5 – Running the Windows-based hello-world container

The only difference on this occasion is that rather than the Linux image, Docker pulled the `windows-amd64` version of the image that is based on the `nanoserver-sac2016` image.

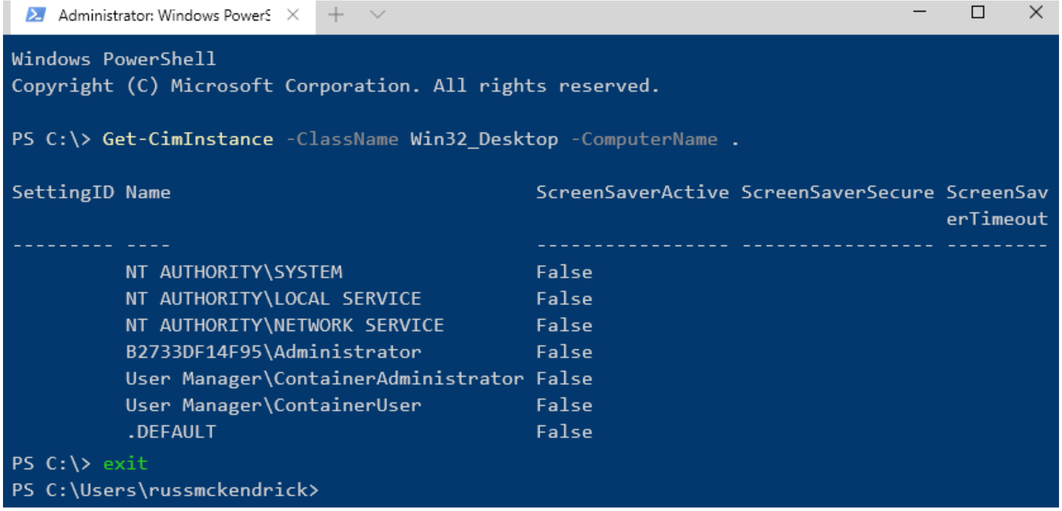
Now, let's look at running a container in the foreground, this time running PowerShell, as follows:

```
$ docker pull mcr.microsoft.com/windows/servercore
$ docker container run -it mcr.microsoft.com/windows/
servercore:ltsc2019 powershell
```

Once your shell is active, running the following command will give you the computer name, which is the container ID:

```
$ Get-CimInstance -ClassName Win32_Desktop -ComputerName .
```

You can see the full Terminal output of the preceding commands in the following screenshot:



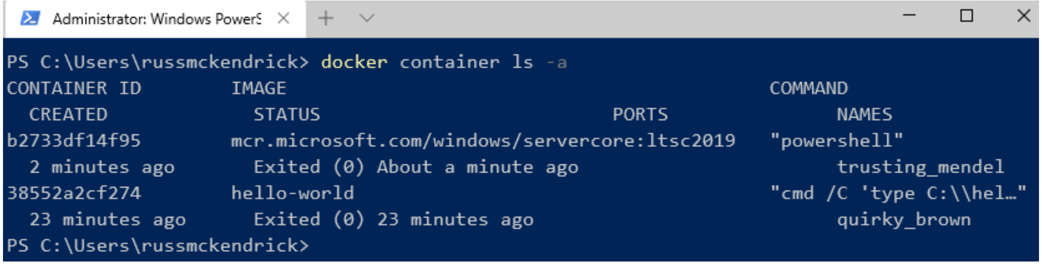
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\> Get-CimInstance -ClassName Win32_Desktop -ComputerName .

SettingID Name                              ScreenSaverActive ScreenSaverSecure ScreenSav
                                                                                erTimeout
--------- ----                              ----------------- ----------------- ---------
          NT AUTHORITY\SYSTEM               False
          NT AUTHORITY\LOCAL SERVICE        False
          NT AUTHORITY\NETWORK SERVICE      False
          B2733DF14F95\Administrator        False
          User Manager\ContainerAdministrator False
          User Manager\ContainerUser        False
          .DEFAULT                          False
PS C:\> exit
PS C:\Users\russmckendrick>
```

Figure 7.6 – Running PowerShell within a container

Once you have exited PowerShell by running `exit`, you can see the container ID by running the following command:

```
$ docker container ls -a
```

You can see the expected output in the following screenshot:



```
PS C:\Users\russmckendrick> docker container ls -a
CONTAINER ID        IMAGE                                              COMMAND
  CREATED             STATUS                          PORTS                NAMES
b2733df14f95        mcr.microsoft.com/windows/servercore:ltsc2019   "powershell"
  2 minutes ago       Exited (0) About a minute ago                        trusting_mendel
38552a2cf274        hello-world                                     "cmd /C 'type C:\\hel…"
  23 minutes ago      Exited (0) 23 minutes ago                            quirky_brown
PS C:\Users\russmckendrick>
```

Figure 7.7 – Checking the containers

Now, let's take a look at building an image that does something a little more adventurous than run PowerShell—instead, let's install a web server.

A Windows container Dockerfile

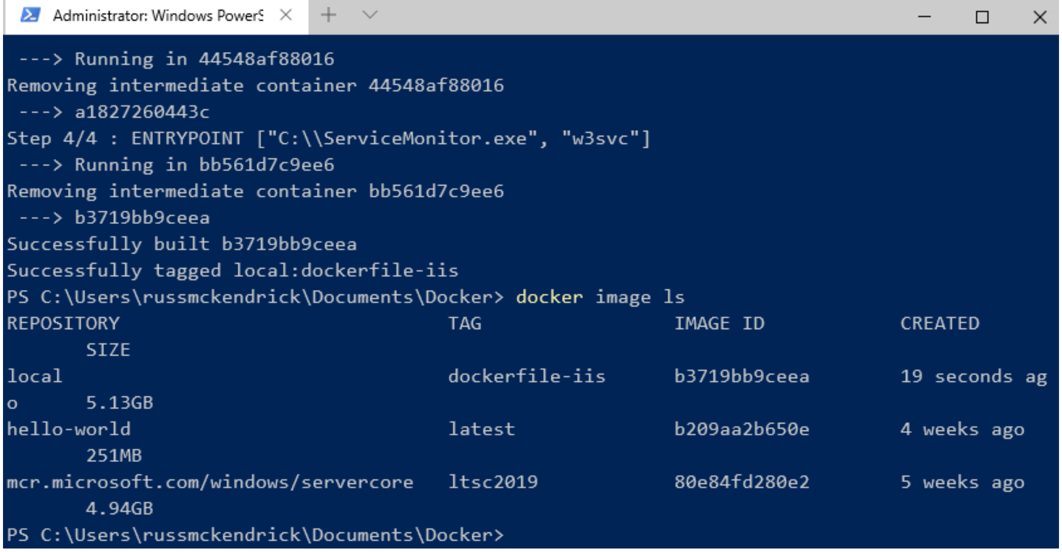
Windows container images use Dockerfile commands in the same format as for Linux containers. The following Dockerfile will download, install, and enable the **Internet Information Services** (**IIS**) web server on the container:

```
# escape=`
FROM mcr.microsoft.com/windows/servercore:ltsc2019

RUN powershell -Command `
    Add-WindowsFeature Web-Server; `
    Invoke-WebRequest -UseBasicParsing -Uri “https://
dotnetbinaries.blob.core.windows.net/servicemonitor/2.0.1.10/
ServiceMonitor.exe” -OutFile “C:\ServiceMonitor.exe”

EXPOSE 80

ENTRYPOINT [“C:\\ServiceMonitor.exe”, “w3svc”]
```

You can build the image using the following command:

```
$ docker image build --tag local:dockerfile-iis .
```

Once built, running `docker image ls` should show you the following:



```
 ---> Running in 44548af88016
Removing intermediate container 44548af88016
 ---> a1827260443c
Step 4/4 : ENTRYPOINT ["C:\\ServiceMonitor.exe", "w3svc"]
 ---> Running in bb561d7c9ee6
Removing intermediate container bb561d7c9ee6
 ---> b3719bb9ceea
Successfully built b3719bb9ceea
Successfully tagged local:dockerfile-iis
PS C:\Users\russmckendrick\Documents\Docker> docker image ls
REPOSITORY                             TAG               IMAGE ID          CREATED
       SIZE
local                                  dockerfile-iis    b3719bb9ceea      19 seconds ag
o      5.13GB
hello-world                            latest            b209aa2b650e      4 weeks ago
       251MB
mcr.microsoft.com/windows/servercore   ltsc2019          80e84fd280e2      5 weeks ago
       4.94GB
PS C:\Users\russmckendrick\Documents\Docker>
```

Figure 7.8 – Building and listing our image

The one immediate thing you will notice about Windows container images is that they are big. Running the container with the following command will start the IIS image:

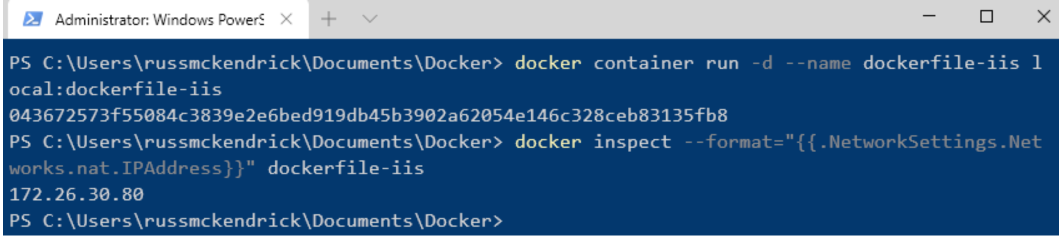
```
$ docker container run -d --name dockerfile-iis
local:dockerfile-iis
```

You can see your newly launched container in action by opening your browser. However, going to `http://localhost:8080/` won't work as we have not provided any ports. If you remember, we are running Docker on Windows, so the containers are running directly on the host machine. Thus, there is no need to use localhost or mapped ports as we can access the container's **network address translation** (NAT) **Internet Protocol** (IP) directly on the host machine of the container.

To find the NAT IP address, you can use the following command:

```
$ docker container inspect --format="{{.NetworkSettings.
Networks.nat.IPAddress}}" dockerfile-iis
```

This should output something like the following:



```
PS C:\Users\russmckendrick\Documents\Docker> docker container run -d --name dockerfile-iis l
ocal:dockerfile-iis
043672573f55084c3839e2e6bed919db45b3902a62054e146c328ceb83135fb8
PS C:\Users\russmckendrick\Documents\Docker> docker inspect --format="{{.NetworkSettings.Net
works.nat.IPAddress}}" dockerfile-iis
172.26.30.80
PS C:\Users\russmckendrick\Documents\Docker>
```

Figure 7.9 – Running our Windows container image

This will give you an IP address. To access IIS, we simply need to put the IP address into a browser running on the Windows host. In this case, the **Uniform Resource Locator** (**URL**) I used was `http://172.26.30.80/`. You should see the following default holding page:

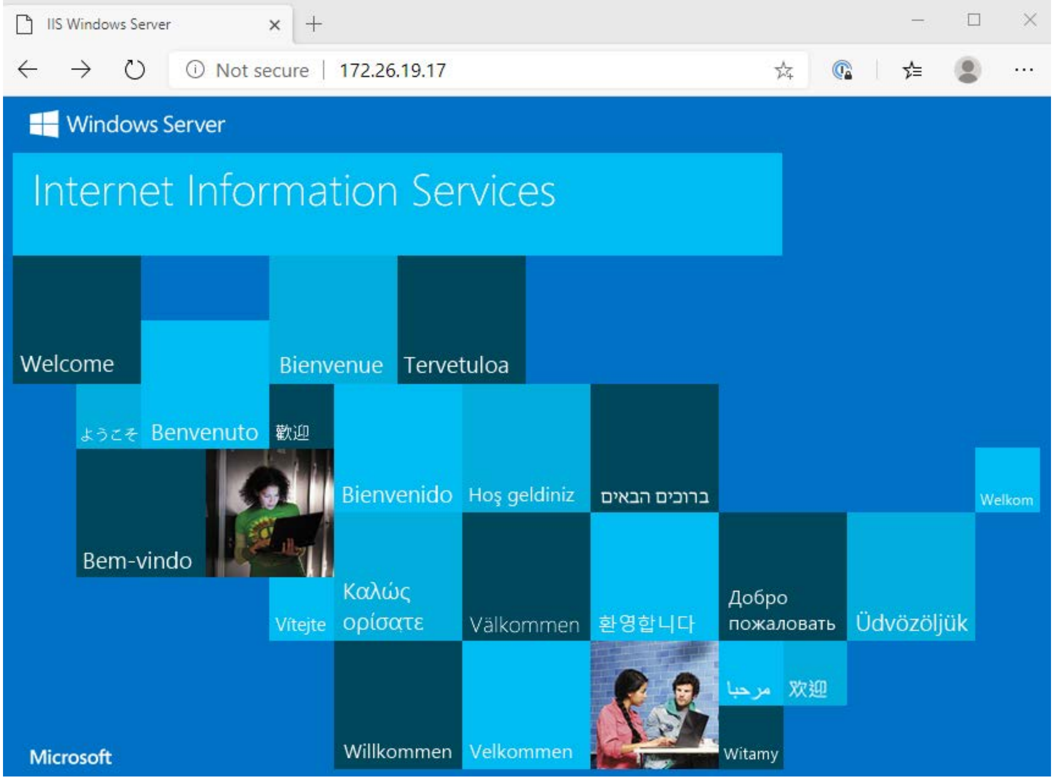


Figure 7.10 – The IIS web server running in a container

To stop and remove the containers we have launched so far, run the following commands:

```
$ docker container stop dockerfile-iis
$ docker container prune
```

So far, I am sure you will agree that the experience is no different from using Docker with Linux-based containers.

Windows containers and Docker Compose

In the final section of this chapter, we are going to look at using Docker Compose with our Windows Docker host. As you will have already guessed, there isn't much change from the commands we ran in *Chapter 5*, *Docker Compose*.

In the `chapter07` folder in the repository, you will find a `docker-compose.yml` file that looks like the following:

```
version: '2.1'

services:
  db:
    image: microsoft/mssql-server-windows-express
    environment:
      sa_password: "${SA_PASSWORD}"
      ACCEPT_EULA: "${SQL_SERVER_ACCEPT_EULA}"
    healthcheck:
      test: [ "CMD", "sqlcmd", "-U", "sa", "-P", "${SA_
PASSWORD}", "-Q", "select 1" ]
      interval: 10s
      retries: 10
  octopus:
    image: octopusdeploy/octopusdeploy:${OCTOPUS_VERSION}
    environment:
      ADMIN_USERNAME: "${OCTOPUS_ADMIN_USERNAME}"
      ADMIN_PASSWORD: "${OCTOPUS_ADMIN_PASSWORD}"
      SQLDBCONNECTIONSTRING: "${DB_CONNECTION_STRING}"
      ACCEPT_EULA: "${OCTOPUS_ACCEPT_EULA}"
      ADMIN_EMAIL: "${ADMIN_EMAIL}"
    ports:
     - “1322:8080”
    depends_on:
      db:
        condition: service_healthy
    stdin_open: true
    volumes:
      - “./Repository:C:/Repository”
      - “./TaskLogs:C:/TaskLogs”

networks:
  default:
    external:
      name: nat
```

There is also a supporting `.env` file—this is used by Docker Compose to populate variables in the Docker Compose file and should be placed in the same folder as the `docker-compose.yml` file, as follows:

```
SA_PASSWORD=N0tS3cr3t!
OCTOPUS_VERSION=2019.13.4
DB_CONNECTION_STRING=Server=db,1433;Initial
Catalog=Octopus;Persist Security Info=False;User
ID=sa;Password=N0tS3cr3t!;MultipleActiveResultSets=False;
Connection Timeout=30;
OCTOPUS_ADMIN_USERNAME=admin
OCTOPUS_ADMIN_PASSWORD=Passw0rd123
ADMIN_EMAIL=
OCTOPUS_ACCEPT_EULA=Y
SQL_SERVER_ACCEPT_EULA=Y
```

As you can see, it is using the same structure, flags, and commands as the previous Docker Compose files we have looked at, the only difference being that we are using images from the Docker Hub that are designed for Windows containers. The Docker Compose file will download Microsoft **Structured Query Language** (**SQL**) `database` and Octopus Deploy (`octopus`). To pull the required images, simply run the following command:

```
$ docker-compose pull
```

Then, once pulled, we need to create the folders required to launch Octopus Deploy, using the following commands in the same folder as the `docker-compose.yml` file:

```
$ mkdir Repository
$ mkdir TaskLogs
```

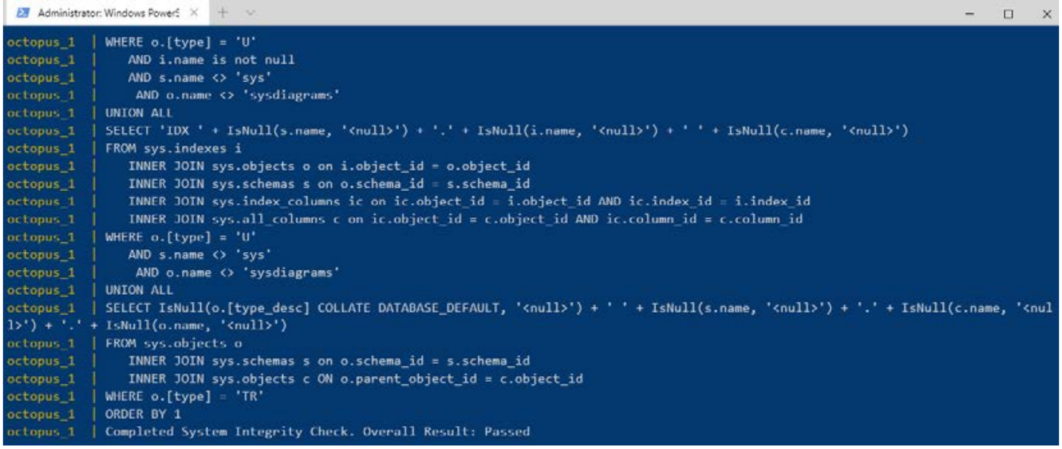
Finally, we can launch Octopus Deploy by running the following command:

```
$ docker-compose --project-name Octopus up -d
```

Octopus can take around 10 minutes to start up. I would recommend running the following command to watch the container logs, to be certain that Octopus Deploy is up and running:

```
$ docker-compose --project-name Octopus logs -f
```

You should see the message `Completed System Integrity Check. Overall Result: Passed` in the log output, which should look something similar to the following output:



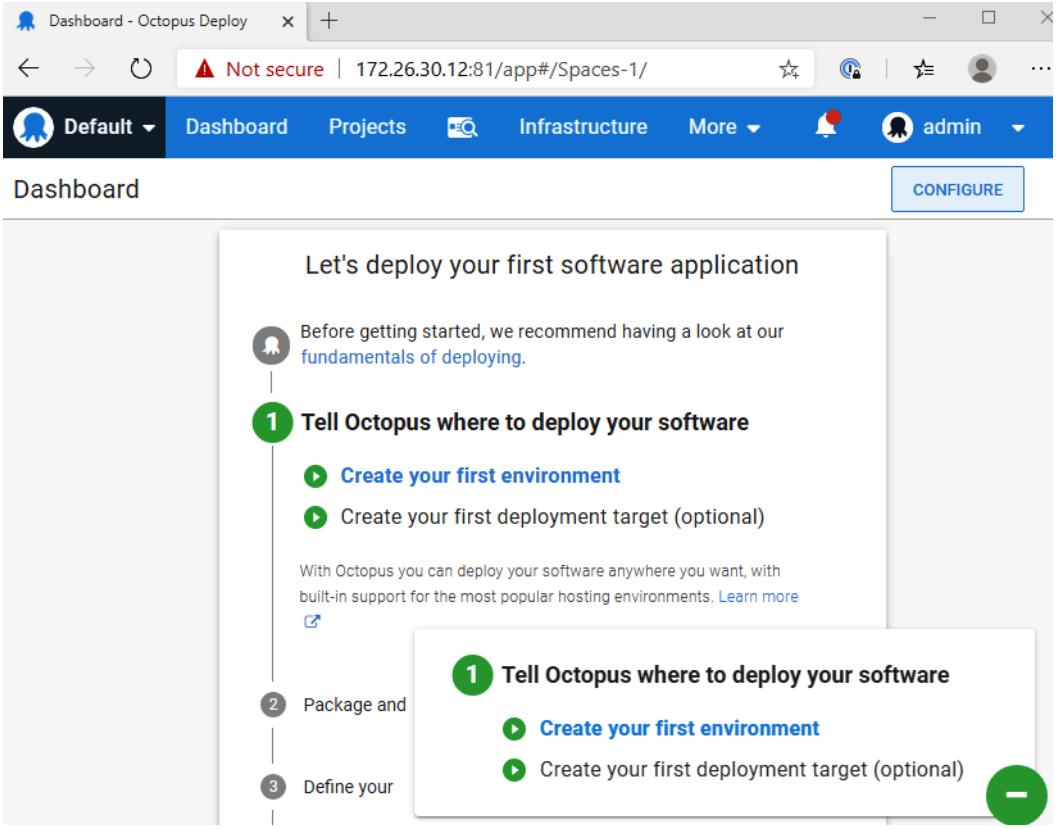
```
octopus_1  | WHERE o.[type] = 'U'
octopus_1  |    AND i.name is not null
octopus_1  |    AND s.name <> 'sys'
octopus_1  |     AND o.name <> 'sysdiagrams'
octopus_1  | UNION ALL
octopus_1  | SELECT 'IDX ' + IsNull(s.name, '<null>') + '.' + IsNull(i.name, '<null>') + ' ' + IsNull(c.name, '<null>')
octopus_1  | FROM sys.indexes i
octopus_1  |    INNER JOIN sys.objects o on i.object_id = o.object_id
octopus_1  |    INNER JOIN sys.schemas s on o.schema_id = s.schema_id
octopus_1  |    INNER JOIN sys.index_columns ic on ic.object_id = i.object_id AND ic.index_id = i.index_id
octopus_1  |    INNER JOIN sys.all_columns c on ic.object_id = c.object_id AND ic.column_id = c.column_id
octopus_1  | WHERE o.[type] = 'U'
octopus_1  |    AND s.name <> 'sys'
octopus_1  |     AND o.name <> 'sysdiagrams'
octopus_1  | UNION ALL
octopus_1  | SELECT IsNull(o.[type_desc] COLLATE DATABASE_DEFAULT, '<null>') + ' ' + IsNull(s.name, '<null>') + '.' + IsNull(c.name, '<nul
l>') + '.' + IsNull(o.name, '<null>')
octopus_1  | FROM sys.objects o
octopus_1  |    INNER JOIN sys.schemas s on o.schema_id = s.schema_id
octopus_1  |    INNER JOIN sys.objects c ON o.parent_object_id = c.object_id
octopus_1  | WHERE o.[type] = 'TR'
octopus_1  | ORDER BY 1
octopus_1  | Completed System Integrity Check. Overall Result: Passed
```

Figure 7.11 – Watching the output of our containers

As before, you can then use this command to find out the IP address on Windows:

```
$ docker inspect -f "{{ .NetworkSettings.Networks.nat.IPAddress
}}" octopus_octopus_1
```

Once you have the IP address, which for me was `172.26.30.12`, open a browser and access the admin interface on port `81`. For me, that was `http://172.26.30.12:81/`. This should show you a login prompt—here, enter the username `admin`, with a password of `Passw0rd123`. Once logged in, you should see something like the following:



Let's deploy your first software application

Before getting started, we recommend having a look at our fundamentals of deploying.

1 Tell Octopus where to deploy your software

Create your first environment

Create your first deployment target (optional)

With Octopus you can deploy your software anywhere you want, with built-in support for the most popular hosting environments. Learn more

2 Package and

3 Define your

1 Tell Octopus where to deploy your software

Create your first environment

Create your first deployment target (optional)

Figure 7.12 – Octopus Deploy up and running

When you are ready, you can remove the containers by running the following command:

```
$ docker-compose --project-name Octopus down --rmi all
--volumes
```

Before we finish, there are a few things to note—the first is the use of a `.env` file. As already mentioned, this saves us having to use hard variables into our Docker Compose files; so, if you ever use them, don't check them into a Git repo like I have done. Secondly, you may have noticed that when we ran the `docker-compose` command, we passed the `--project-name Octopus` parameter. This means that when we launch our project, rather than the application inheriting the name of the folder where the Docker Compose file is, it will be prefixed with `octopus`.

Summary

In this chapter, we have briefly looked at Windows containers. As you have seen, thanks to Microsoft's adoption of Docker as a management tool for Windows containers, the experience is familiar to anyone who has used Docker to manage Linux containers.

In the next chapter, we are going to take a look at Docker Swarm. This will be the first time we have moved from single Docker hosts to a cluster of hosts.

Questions

1. Docker on Windows introduces which additional layer of isolation?
2. Which command would you use to find out the NAT IP address of your Windows container?
3. True or false: Docker on Windows introduces an additional set of commands you need to use in order to manage your Windows containers.

Further reading

You can find more information on the topics mentioned in this chapter, as follows:

- Docker and Microsoft Partnership Announcement: `https://blog.docker.com/2014/10/docker-microsoft-partner-distributed-applications/`
- Windows Server and Docker – The Internals Behind Bringing Docker & Containers to Windows: `https://www.youtube.com/watch?v=85nCF5S8Qok`
- Stefan Scherer on GitHub: `https://github.com/stefanScherer/`
- Octopus Deploy: `https://octopus.com`

8 Clustering with Docker Swarm

In this chapter, we will be taking a look at Docker Swarm. With Docker Swarm, you can create and manage Docker clusters. Swarm can be used to distribute containers across multiple hosts and also has the ability to scale containers.

We will cover the following topics:

- Introducing Docker Swarm
- Creating and managing a swarm
- Docker Swarm services and stacks
- Load balancing, overlays, and scheduling

Technical requirements

As in previous chapters, we will continue to use our local Docker installations. Again, the screenshots in this chapter will be from my preferred operating system, macOS. As before, the Docker commands we will be running will work on all three of the operating systems on which we have installed Docker so far. However, some of the supporting commands, which will be few and far between, may only apply to macOS- and Linux-based operating systems.

Check out the following video to see the Code in Action:
`https://bit.ly/334RE0A`

Introducing Docker Swarm

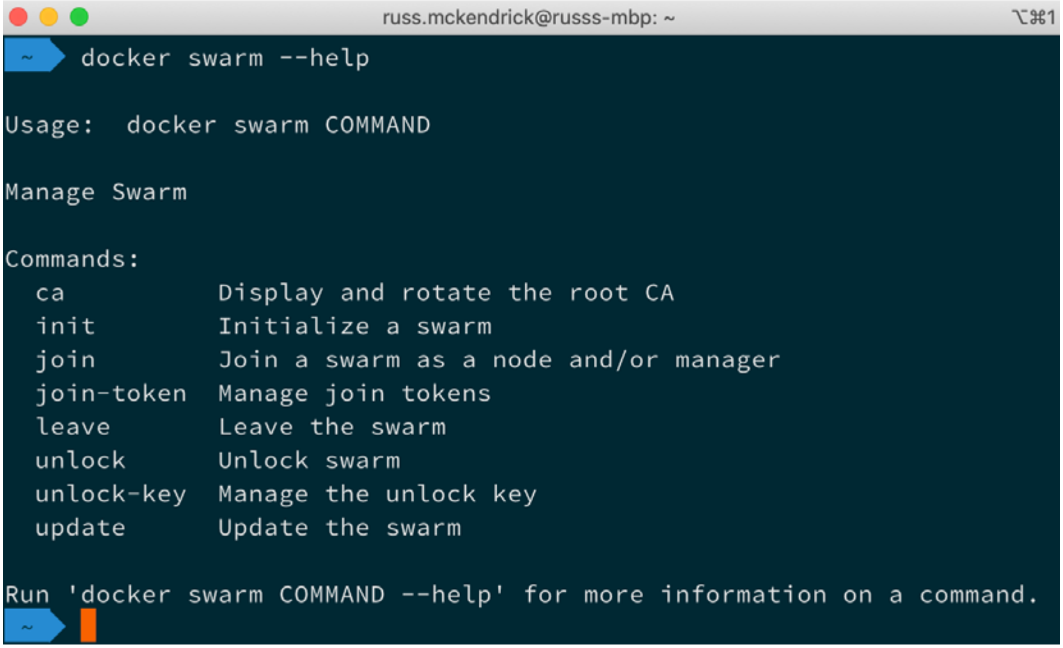
Before we go any further, I should mention that there are two very different versions of Docker Swarm. There was a standalone version of Docker Swarm—this was supported up until Docker `1.12` and is no longer being actively developed; however, you may find some old documentation mentions it. Installation of the standalone Docker Swarm is not recommended as Docker ended support for version `1.11.x` in the first quarter of 2017.

Docker version `1.12` introduced Docker Swarm mode. This introduced all of the functionality that was available in the standalone Docker Swarm version into the core Docker Engine, along with a significant number of additional features. As we are covering Docker 19.03 and higher in this book, we will be using Docker Swarm mode, which, for the remainder of the chapter, we will refer to as Docker Swarm.

As you are already running a version of Docker with in-built support for Docker Swarm, there isn't anything you need to do in order to install Docker Swarm. You can verify that Docker Swarm is available on your installation by running the following command:

```
$ docker swarm --help
```

You should see something that looks like the following Terminal output when running the command:



```
~ docker swarm --help

Usage:  docker swarm COMMAND

Manage Swarm

Commands:
  ca          Display and rotate the root CA
  init        Initialize a swarm
  join        Join a swarm as a node and/or manager
  join-token  Manage join tokens
  leave       Leave the swarm
  unlock      Unlock swarm
  unlock-key  Manage the unlock key
  update      Update the swarm

Run 'docker swarm COMMAND --help' for more information on a command.
~
```

Figure 8.1 – Viewing the help

If you get an error, ensure that you are running Docker 19.03 or higher, the installation of which we covered in *Chapter 1*, *Docker Overview*. Now that we know that our Docker client supports Docker Swarm, what do we mean by a Swarm?

A **Swarm** is a collection of hosts, all running Docker, which have been set up to interact with each other in a clustered configuration. Once configured, you will be able to use all of the commands we have been running so far when targeting a single host, and let Docker Swarm decide the placement of your containers by using a deployment strategy to decide the most appropriate host on which to launch your container. Docker Swarms are made up of two types of host. Let's take a look at these now.

Roles within a Docker Swarm cluster

Which roles are involved with Docker Swarm? Let's take a look at the two roles a host can assume when running within a Docker Swarm cluster.

Swarm manager

The **Swarm manager** is a host that is the central management point for all Swarm hosts. The Swarm manager is where you issue all your commands to control those nodes. You can switch between the nodes, join nodes, remove nodes, and manipulate those hosts.

Each cluster can run several Swarm managers. For production, it is recommended that you run a minimum of five Swarm managers; this would mean that our cluster can take a maximum of two Swarm manager node failures before you start to encounter any errors. Swarm managers use the *Raft consensus algorithm* (see the *Further reading* section for more details) to maintain a consistent state across all of the manager nodes.

Swarm workers

The Swarm workers, which we have seen referred to earlier as Docker hosts, are those that run the Docker containers. Swarm workers are managed from the Swarm manager, and are depicted in the following diagram:

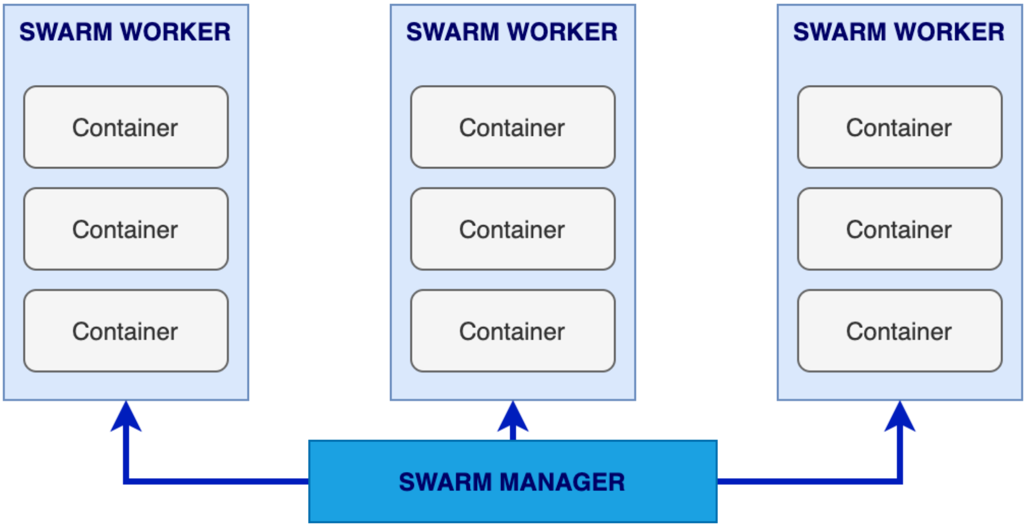


Figure 8.2 – An overview of Swarm workers

This is an illustration of all the Docker Swarm components. We see that the Docker Swarm manager talks to each Swarm host that has a Docker Swarm worker role. The workers do have some level of connectivity, which we will look at shortly.

Creating and managing a Swarm

Let's now take a look at using Swarm and how we can perform the following tasks:

- Creating a cluster
- Joining workers
- Listing nodes
- Managing a cluster

Creating the cluster hosts

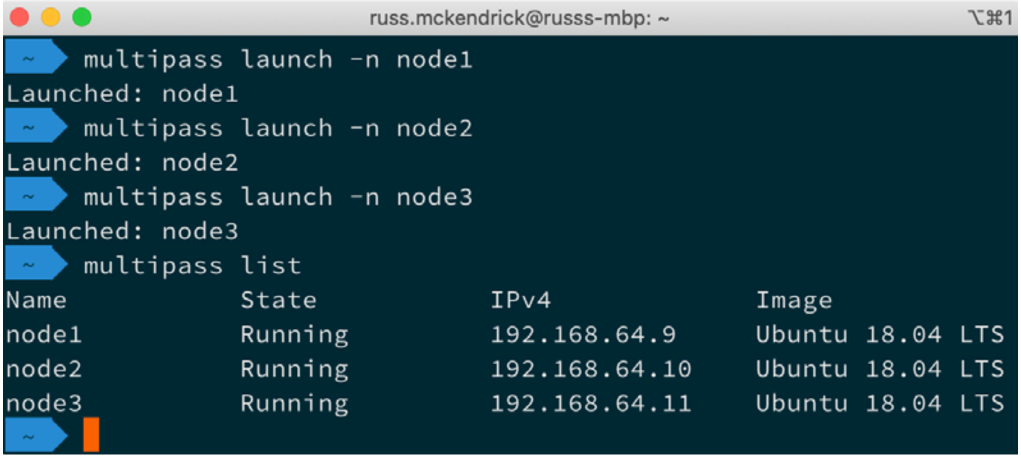
Let's start by creating a cluster of three machines. Since we are going to be creating a multi-node cluster on our local machine, we are going to use Multipass, which we covered in *Chapter 6, Docker Machine, Vagrant, and Multipass*, to launch the hosts by running the following commands:

```
$ multipass launch -n node1
$ multipass launch -n node2
$ multipass launch -n node3
```

This should give us three nodes; you can check this by just running the following command:

```
$ multipass list
```

You should see something similar to the following output:



```
~ multipass launch -n node1
Launched: node1
~ multipass launch -n node2
Launched: node2
~ multipass launch -n node3
Launched: node3
~ multipass list
Name            State           IPv4            Image
node1           Running         192.168.64.9    Ubuntu 18.04 LTS
node2           Running         192.168.64.10   Ubuntu 18.04 LTS
node3           Running         192.168.64.11   Ubuntu 18.04 LTS
~
```

Figure 8.3 – Launching the nodes using Multipass

You may remember from when we last used Multipass that bringing up a host doesn't mean that Docker is installed; so, now, let's install Docker and add the `ubuntu` user to the `docker` group so that when we use `multipass exec`, we don't have to change user. To do this, run the following three commands:

```
$ multipass exec node1 -- \
    /bin/bash -c 'curl -s https://get.docker.com | sh - && sudo usermod -aG docker ubuntu'

$ multipass exec node2 -- \
    /bin/bash -c 'curl -s https://get.docker.com | sh - && sudo usermod -aG docker ubuntu'

$ multipass exec node3 -- \
    /bin/bash -c 'curl -s https://get.docker.com | sh - && sudo usermod -aG docker ubuntu'
```

Now that we have our three cluster nodes ready, we can move on to the next step, which is adding a Swarm manager to the cluster.

Adding a Swarm manager to the cluster

Let's bootstrap our Swarm manager. To do this, we will pass the results of a few Docker Machine commands to our host. Before we create the Swarm manager, we need to get the IP address of `node1` as this is going to be our Swarm manager. If you are using macOS or Linux, then you can set an environment variable by running the following command:

```
$ IP=$(multipass info node1 | grep IPv4 | awk '{print $2}')
```

If you are using Windows 10, then run `multipass list` and make a note of the IP address for `node1`.

The command to run in order to create our manager is shown in the following code snippet (if you are running Windows, replace `$IP` with the IP address you made a note of):

```
$ multipass exec node1 -- \
    /bin/bash -c 'docker swarm init --advertise-addr $IP:2377 --listen-addr $IP:2377'
```

You should receive a message similar to this one:

```
Swarm initialized: current node (92kts1c9x17gbqv3in9t1w4qm) is
now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-4s8vpkileg2l2sicpya
y5fojhis9jygb8mv04tsy9jmeqmzhk8-4cp4hp0i1qjqpuln6q3ytprtc
192.168.64.9:2377

To add a manager to this swarm, run 'docker swarm join-token
manager' and follow the instructions.
```

As you can see from the output, once your manager is initialized, you are given a unique token.

In the preceding example, the full token is this:

```
SWMTKN-1-4s8vpkileg2l2sicpyay5fojhis9jygb8mv04tsy9jmeqmzhk8-
4cp4hp0i1qjqpuln6q3ytprtc.
```

This token will be needed for the worker nodes to authenticate themselves and join our cluster.

Joining Swarm workers to the cluster

Now it is time to add our two workers (`node2` and `node3`) to the cluster. First, let's set an environment variable to hold our token, making sure that you replace the token with the one you received when initializing your own manager, as follows:

```
$ SWARM_TOKEN=$(multipass exec node1 -- /bin/bash -c 'docker
swarm join-token --quiet worker')
```

Again, Windows users need to make a note of the token and will have to replace both `$SWARM_TOKEN` and `$IP` in the command shown next with their respective values.

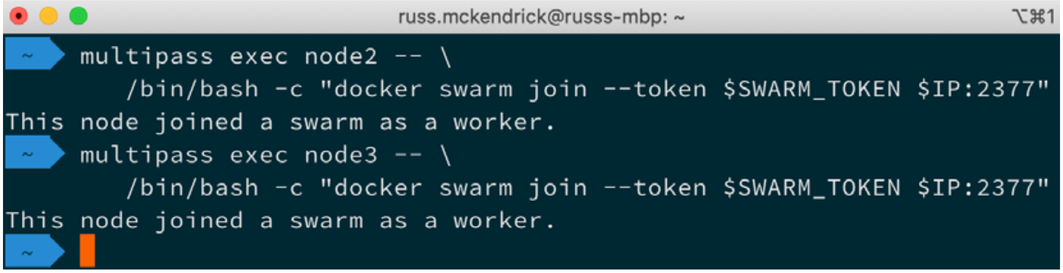
Now, we can run the following command to add `node2` to the cluster:

```
$ multipass exec node2 -- \
    /bin/bash -c 'docker swarm join --token $SWARM_TOKEN 
$IP:2377'
```

For `node3`, you need to run the following command:

```
$ multipass exec node3 -- \
    /bin/bash -c 'docker swarm join --token $SWARM_TOKEN 
$IP:2377'
```

Both times, you should get confirmation that your node has joined the cluster, as illustrated in the following screenshot:



```
~ multipass exec node2 -- \
     /bin/bash -c "docker swarm join --token $SWARM_TOKEN $IP:2377"
This node joined a swarm as a worker.
~ multipass exec node3 -- \
     /bin/bash -c "docker swarm join --token $SWARM_TOKEN $IP:2377"
This node joined a swarm as a worker.
~
```

Figure 8.4 – Adding the workers to the cluster

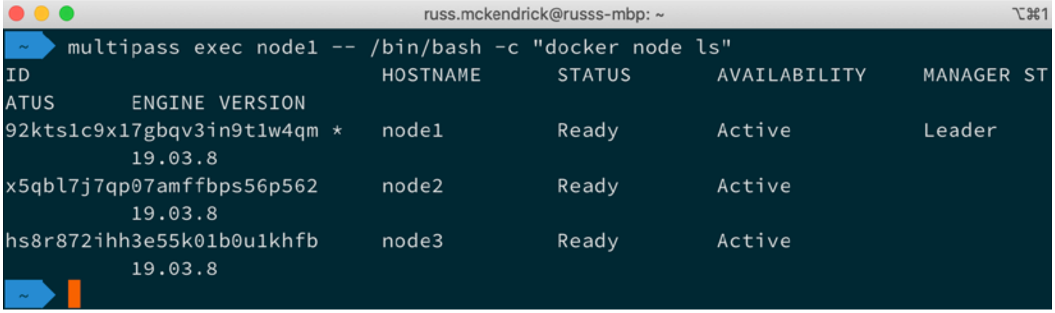
Listing nodes

You can check the Swarm by running the following command:

```
$ multipass exec node1 -- /bin/bash -c 'docker node ls'
```

This will connect to `node1`, which we have configured as the Swarm master, and query all of the nodes that form our cluster.

You should see that all three of our nodes are listed, as illustrated in the following screenshot:



```
~ multipass exec node1 -- /bin/bash -c "docker node ls"
ID                          HOSTNAME      STATUS      AVAILABILITY     MANAGER ST
ATUS      ENGINE VERSION
92kts1c9x17gbqv3in9t1w4qm *   node1         Ready       Active           Leader
          19.03.8
x5qbl7j7qp07amffbps56p562     node2         Ready       Active
          19.03.8
hs8r872ihh3e55k01b0u1khfb     node3         Ready       Active
          19.03.8
~
```

Figure 8.5 – Listing the cluster nodes

Now, we are going to move from the Docker client on our local machine to that on node1. To connect to the shell on node1, we just need to run the following command:

```
$ multipass shell node1
```

This will leave us at a prompt on node1, where we are ready to start using our cluster.

Managing a cluster

Let's see how we can perform some management of all of these cluster nodes that we are creating.

There are only two ways in which you can go about managing the containers within your cluster—these are by using the docker service and docker stack commands, which we are going to be covering in the next section of the chapter.

Before we look at launching containers in our cluster, let's have a look at managing the cluster itself, starting with how you can find out more information on it.

Finding information on the cluster

As we have already seen, we can list the nodes within the cluster using the Docker client installed on node1. To find out more information, we can simply type this to the command line of node1:

```
$ docker info
```

This will give us lots of information about the host, as you can see from the following output, which I have truncated:

```
Server:
 Containers: 0
 Images: 0
 Server Version: 19.03.8
 Swarm: active
  NodeID: 92kts1c9x17gbqv3in9t1w4qm
  Is Manager: true
  ClusterID: k65y4ke5rmup1n74z9lb9gerx
  Managers: 1
  Nodes: 3
  Default Address Pool: 10.0.0.0/8
  SubnetSize: 24
```

As you can see, there is information about the cluster in the `Swarm` section; however, we are only able to run the `docker info` command against the host with which our client is currently configured to communicate. Luckily, the `docker node` command is cluster-aware, so we can use that to get information on each node within our cluster.

> **TIP**
>
> Assessing the `--pretty` flag with the `docker node inspect` command will render the output in the easy-to-read format you see next. If `--pretty` is left out, Docker will return the raw `JSON` object containing the results of the query the `inspect` command runs against the cluster.

Here is what we would need to run to get information on `node1`:

```
$ docker node inspect node1 -pretty
```

This should provide the following information on our Swarm manager:

```
ID:              92kts1c9x17gbqv3in9t1w4qm
Hostname:                   node1
Joined at:                  2020-04-12 14:00:02.218330889
+0000 utc
Status:
 State:               Ready
 Availability:              Active
 Address:        192.168.64.9
Manager Status:
 Address:        192.168.64.9:2377
 Raft Status:         Reachable
 Leader:        Yes
Platform:
 Operating System:    linux
 Architecture:        x86_64
Resources:
 CPUs:                1
 Memory:        985.7MiB
Plugins:
 Log:           awslogs, fluentd, gcplogs, gelf, journald,
json-file, local, logentries, splunk, syslog
 Network:       bridge, host, ipvlan, macvlan, null, overlay
 Volume:        local
Engine Version:       19.03.8
```

Run the same command, but this time targeting one of the worker nodes, as follows:

```
$ docker node inspect node2 --pretty
```

This gives us similar information, as can be seen here:

```
ID:             x5qbl7j7qp07amffbps56p562
Hostname:                   node2
Joined at:                  2020-04-12 14:10:15.08034922 +0000
utc
Status:
 State:               Ready
 Availability:              Active
 Address:       192.168.64.10
Platform:
 Operating System:    linux
 Architecture:        x86_64
Resources:
 CPUs:                1
 Memory:        985.7MiB
Plugins:
 Log:           awslogs, fluentd, gcplogs, gelf, journald,
json-file, local, logentries, splunk, syslog
 Network:       bridge, host, ipvlan, macvlan, null, overlay
```

```
  Volume:        local
 Engine Version:       19.03.8
```

But as you can see, information about the state of the manager functionality is missing. This is because the worker nodes do not need to know about the status of the manager nodes; they just need to know that they are allowed to receive instructions from the managers.

In this way, we can see details about this host, such as the number of containers, the number of images on the host, and information about the **central processing unit** (**CPU**) and memory, along with other interesting information.

Now that we know how to get information on the nodes that go to make up our cluster, let's take a look at how we can promote a node's role within the cluster.

Promoting a worker node

Say you wanted to perform some maintenance on your single manager node, but you wanted to maintain the availability of your cluster. No problem—you can promote a worker node to a manager node.

While we have our local three-node cluster up and running, let's promote `node2` to be a new manager. To do this, run the following command:

```
$ docker node promote node2
```

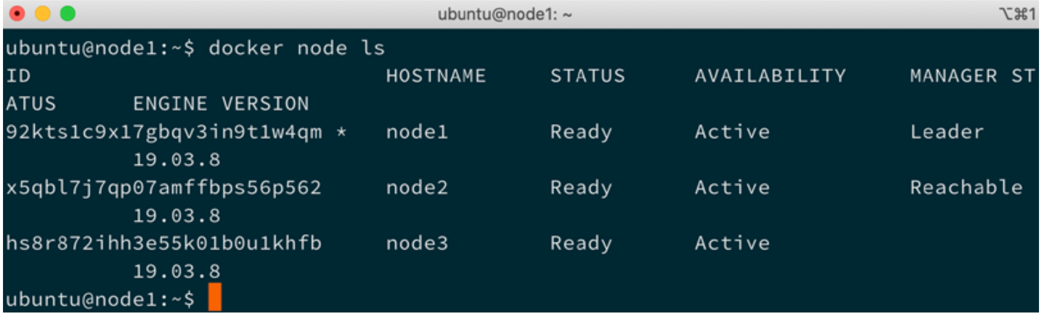
You should receive a message confirming that your node has been promoted immediately after executing the command, as follows:

```
Node node2 promoted to a manager in the swarm.
```

List the nodes by running this command:

```
$ docker node ls
```

This should show you that you now have two nodes that display something in the `MANAGER STATUS` column, as illustrated in the following screenshot:



```
ubuntu@node1:~$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY    MANAGER ST
ATUS      ENGINE VERSION
92kts1c9x17gbqv3in9t1w4qm *   node1      Ready     Active          Leader
          19.03.8
x5qbl7j7qp07amffbps56p562     node2      Ready     Active          Reachable
          19.03.8
hs8r872ihh3e55k01b0u1khfb     node3      Ready     Active
          19.03.8
ubuntu@node1:~$
```

Figure 8.6 – Checking the status of the nodes in the cluster

Our `node1` node is still the primary manager node, though. Let's look at doing something about that, and switch its role from manager to worker.

Demoting a manager node

You may have already put two and two together, but to demote a manager node to a worker node, you simply need to run this command:

```
$ docker node demote node1
```

Again, you will receive immediate feedback stating the following:

```
Manager node1 demoted in the swarm.
```

Now that we have demoted our node, you can check the status of the nodes within the cluster by running this command:

```
$ docker node ls
```

As we are connected to `node1`, which is the newly demoted node, you will receive a message stating the following:

```
Error response from daemon: This node is not a swarm manager.
Worker nodes can't be used to view or modify cluster state.
Please run this command on a manager node or promote the
current node to a manager.
```

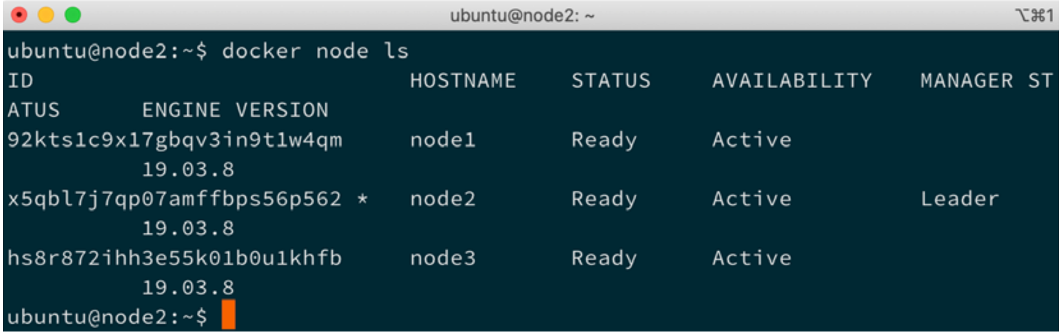
To connect to our new Swarm manager, we need to be SSHd into `node2`. To do this, we simply need to disconnect from `node1` and connect to `node2` by running the following:

```
$ exit
$ multipass shell node2
```

Now that are connected to a manager node again, rerun this, as follows:

```
$ docker node ls
```

It should list the nodes as expected, as illustrated in the following screenshot:



```
ubuntu@node2:~$ docker node ls
ID                            HOSTNAME    STATUS     AVAILABILITY   MANAGER ST
ATUS       ENGINE VERSION
92kts1c9x17gbqv3in9t1w4qm     node1       Ready      Active
          19.03.8
x5qbl7j7qp07amffbps56p562 *   node2       Ready      Active         Leader
          19.03.8
hs8r872ihh3e55k01b0u1khfb     node3       Ready      Active
          19.03.8
ubuntu@node2:~$ 
```

Figure 8.7 – Checking the status of the nodes in the cluster

This is all well and good, but how would we take a node out of the cluster so that we could perform maintenance on it? Let's now take a look at how we would drain a node.

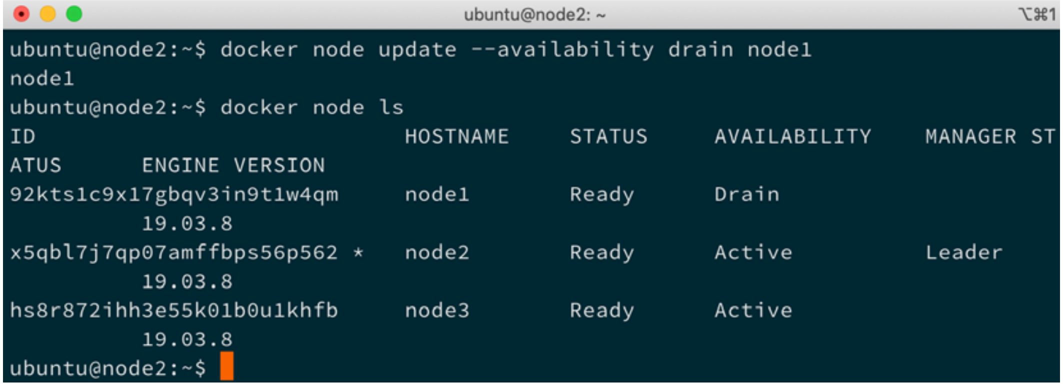
Draining a node

To temporarily remove a node from our cluster so that we can perform maintenance, we need to set the status of the node to `Drain`. Let's look at draining our former manager node. To do this, we need to run the following command:

```
$ docker node update --availability drain node1
```

This will stop any new tasks, such as new containers launching or being executed against the node we are draining. Once new tasks have been blocked, all running tasks will be migrated from the node we are draining to nodes with an `Active` status.

As you can see from the following Terminal output, listing the nodes now shows that `node1` is listed with a status of `Drain` in the `AVAILABILITY` column:



```
ubuntu@node2:~$ docker node update --availability drain node1
node1
ubuntu@node2:~$ docker node ls
ID                            HOSTNAME    STATUS    AVAILABILITY    MANAGER ST
ATUS      ENGINE VERSION
92kts1c9x17gbqv3in9t1w4qm     node1       Ready     Drain
          19.03.8
x5qbl7j7qp07amffbps56p562 *   node2       Ready     Active          Leader
          19.03.8
hs8r872ihh3e55k01b0u1khfb     node3       Ready     Active
          19.03.8
ubuntu@node2:~$
```

Figure 8.8 – Checking the status of our cluster

Now that our node is no longer accepting new tasks and all running tasks have been migrated to our two remaining nodes, we can safely perform our maintenance, such as rebooting the host. To reboot `node1`, run the following two commands on your main host in a second window:

```
$ multipass shell node1
$ sudo reboot
```

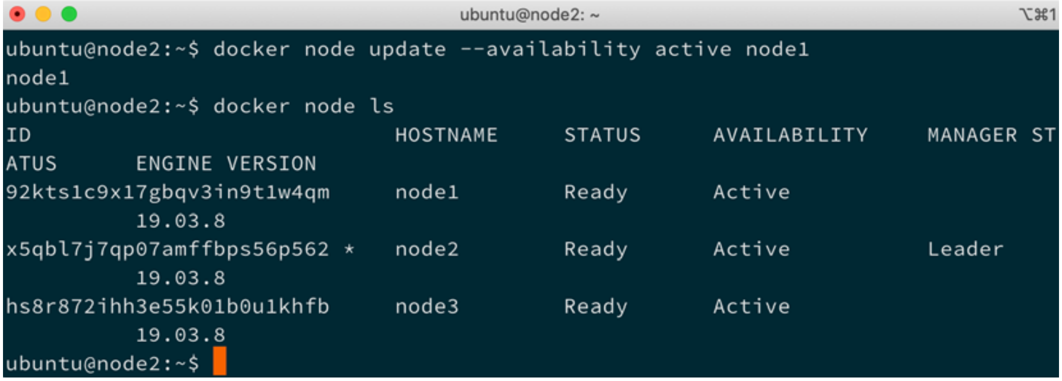
Once the host has been rebooted, run this command on node2:

```
$ docker node ls
```

It should show that the node has an AVAILABILITY status of Drain. To add the node back into the cluster, simply change the AVAILABILITY status to Active by running the following on node2:

```
$ docker node update --availability active node1
```

As you can see from the following Terminal output, our node is now active, meaning new tasks can be executed against it:



```
ubuntu@node2:~$ docker node update --availability active node1
node1
ubuntu@node2:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER ST
ATUS          ENGINE VERSION
92kts1c9x17gbqv3in9t1w4qm     node1               Ready               Active
              19.03.8
x5qbl7j7qp07amffbps56p562 *   node2               Ready               Active              Leader
              19.03.8
hs8r872ihh3e55k01b0u1khfb     node3               Ready               Active
              19.03.8
ubuntu@node2:~$
```

Figure 8.9 – Checking the status of our cluster

Now that we have looked at how to create and manage a Docker Swarm cluster, we should look at how to run a task such as creating and scaling a service or launching a stack.

Docker Swarm services and stacks

So far, we have looked at the following commands:

```
$ docker swarm <command>
$ docker node <command>
```

These two commands allow us to bootstrap and manage our Docker Swarm cluster from a collection of existing Docker hosts. The next two commands we are going to look at are as follows:

```
$ docker service <command>
$ docker stack <command>
```

The `service` and `stack` commands allow us to execute tasks that, in turn, launch, scale, and manage containers within our Swarm cluster.

Services

The `service` command is a way of launching containers that take advantage of the Swarm cluster. Let's look at launching a really basic single-container service on our Swarm cluster.

> **TIP**
>
> Don't forget that the `docker` commands here need to be executed from your current Swarm manager. If you are following, that should be `node2`.

To do this, run the following command:

```
$ docker service create \
    --name cluster \
    --constraint 'node.role == worker' \
    -p:80:80/tcp \
    russmckendrick/cluster
```

This will create a service called `cluster` that consists of a single container with port `80` mapped from the container to the host machine, and it will only be running on nodes that have the role of `worker`.

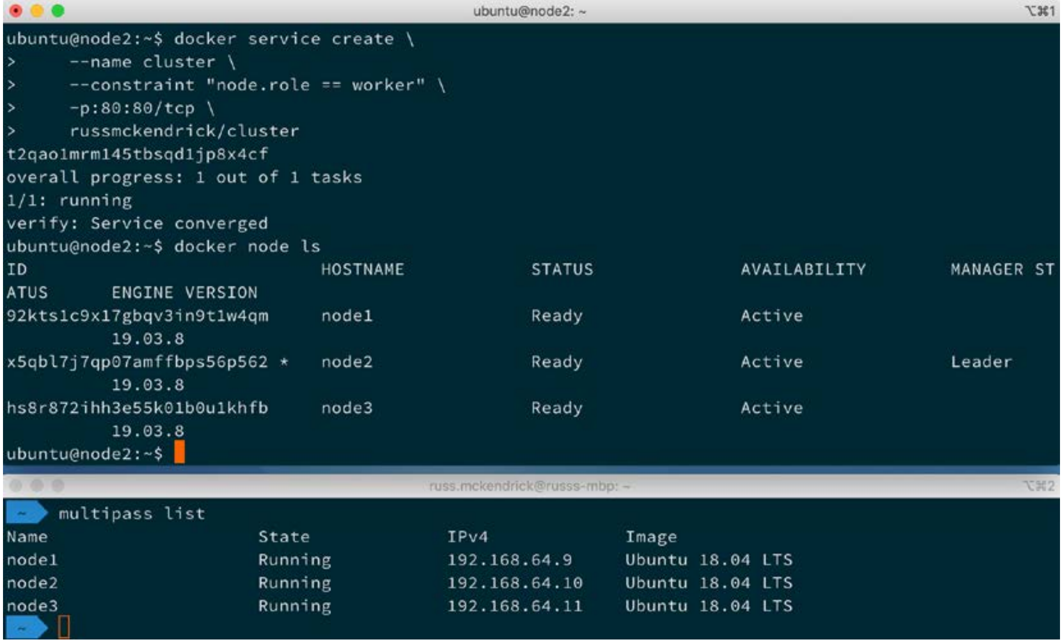
Before we look at doing more with the service, we can check whether it worked on our browser. To do this, we will need the IP address of our two worker nodes. First of all, we need to double-check which are the worker nodes by running this command:

```
$ docker node ls
```

Once we know which node has which role, you can find the IP addresses of your nodes by running this command in a second Terminal window on your host machine:

```
$ multipass list
```

Look at the following Terminal output:



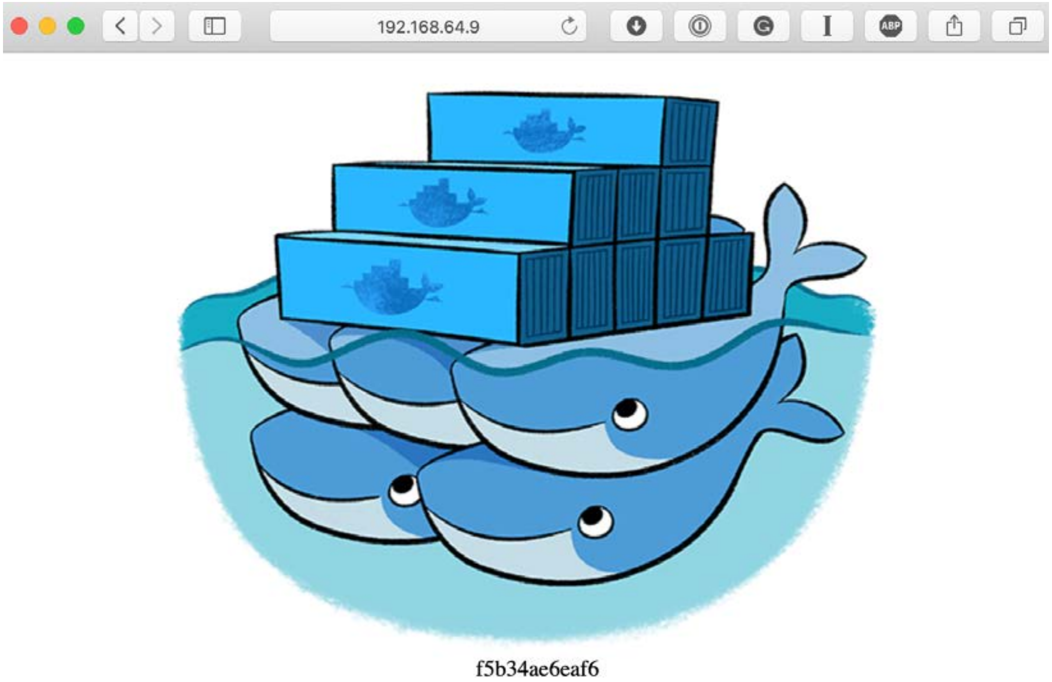
```
ubuntu@node2:~$ docker service create \
>     --name cluster \
>     --constraint "node.role == worker" \
>     -p:80:80/tcp \
>     russmckendrick/cluster
t2qao1mrm145tbsqd1jp8x4cf
overall progress: 1 out of 1 tasks
1/1: running
verify: Service converged
ubuntu@node2:~$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER ST
ATUS      ENGINE VERSION
92kts1c9x17gbqv3in9t1w4qm     node1               Ready               Active
          19.03.8
x5qbl7j7qp07amffbps56p562 *   node2               Ready               Active              Leader
          19.03.8
hs8r872ihh3e55k01b0u1khfb     node3               Ready               Active
          19.03.8
ubuntu@node2:~$
```

```
 ~  multipass list
Name                    State             IPv4             Image
node1                   Running           192.168.64.9     Ubuntu 18.04 LTS
node2                   Running           192.168.64.10    Ubuntu 18.04 LTS
node3                   Running           192.168.64.11    Ubuntu 18.04 LTS
 ~ 
```

Figure 8.10 – Creating a service and checking the IP addresses of the nodes

My worker nodes are `node1` and `node3`, whose IP addresses are `192.168.64.9` and `192.168.64.11`, respectively.

Going to either of the IP addresses of your worker nodes, such as `http://192.168.64.9/` or `http://192.168.64.11/`, in a browser will show the output of the `russmckendrick/cluster` application, which is the Docker Swarm graphic and the hostname of the container the page is being served from. This is illustrated in the following screenshot:



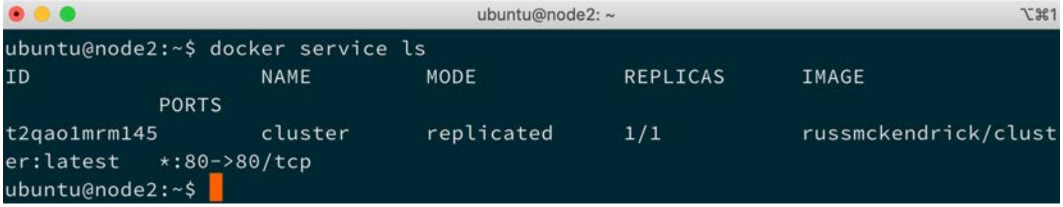
f5b34ae6eaf6

Figure 8.11 – Our cluster application

Now that we have our service running on our cluster, we can start to find out more information about it. First of all, we can list the services again by running this command:

```
$ docker service ls
```

In our case, this should return the single service we launched, called `cluster`, as illustrated in the following screenshot:



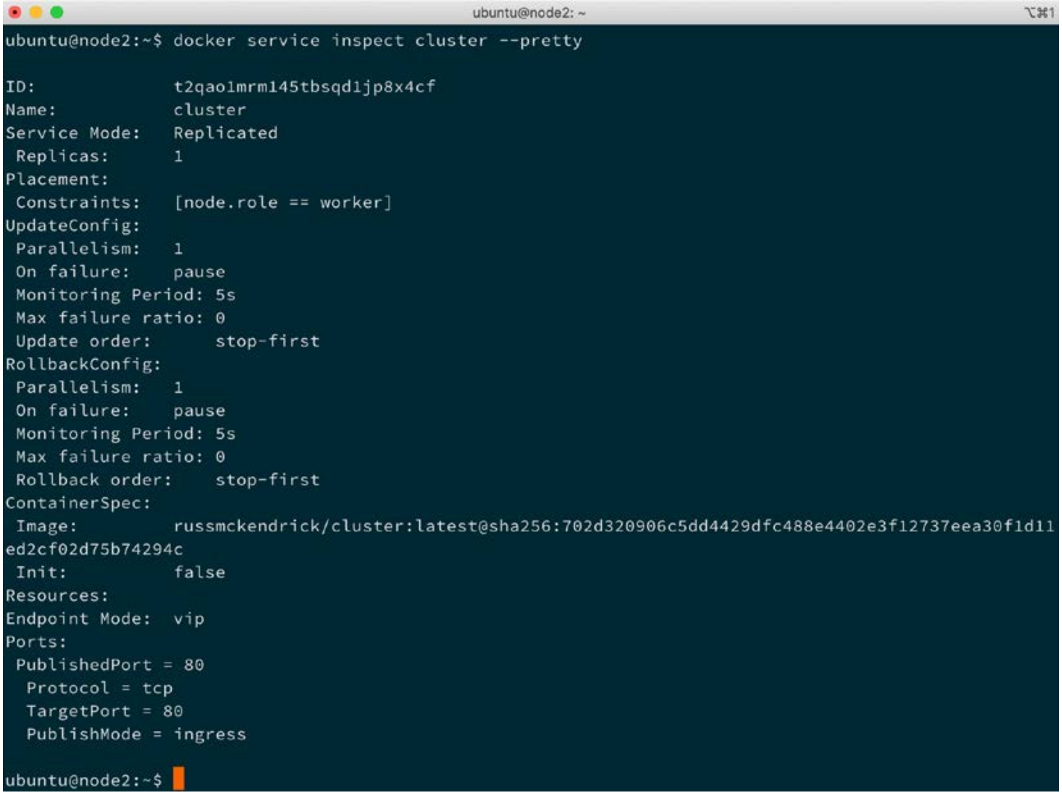
```
ubuntu@node2:~$ docker service ls
ID                  NAME          MODE            REPLICAS       IMAGE
            PORTS
t2qao1mrm145        cluster       replicated      1/1            russmckendrick/clust
er:latest    *:80->80/tcp
ubuntu@node2:~$
```

Figure 8.12 – Listing the services

As you can see, it is a **replicated** service and **1/1** containers are active. Next, you can drill down to find out more information about the service by running the `inspect` command, as follows:

```
$ docker service inspect cluster --pretty
```

This will return detailed information about the service, as illustrated in the following screenshot:



```
ubuntu@node2:~$ docker service inspect cluster --pretty

ID:             t2qao1mrm145tbsqd1jp8x4cf
Name:           cluster
Service Mode:   Replicated
 Replicas:      1
Placement:
 Constraints:   [node.role == worker]
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:         russmckendrick/cluster:latest@sha256:702d320906c5dd4429dfc488e4402e3f12737eea30f1d11
ed2cf02d75b74294c
 Init:          false
Resources:
Endpoint Mode:  vip
Ports:
 PublishedPort = 80
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress

ubuntu@node2:~$
```

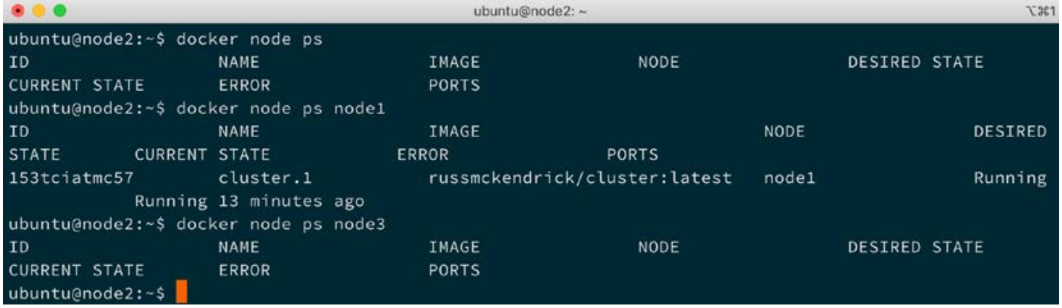
Figure 8.13 – Grabbing information on a service

You may have noticed that so far, we haven't had to bother about which of our two worker nodes the service is currently running on. This is quite an important feature of Docker Swarm, as it completely removes the need for you to worry about the placement of individual containers.

Before we look at scaling our service, we can take a quick look at which host our single container is running on by executing these commands:

```
$ docker node ps
$ docker node ps node1
$ docker node ps node3
```

This will list the containers running on each of our hosts. By default, it will list the host the command is being targeted against, which in my case is `node1`, as illustrated in the following screenshot:



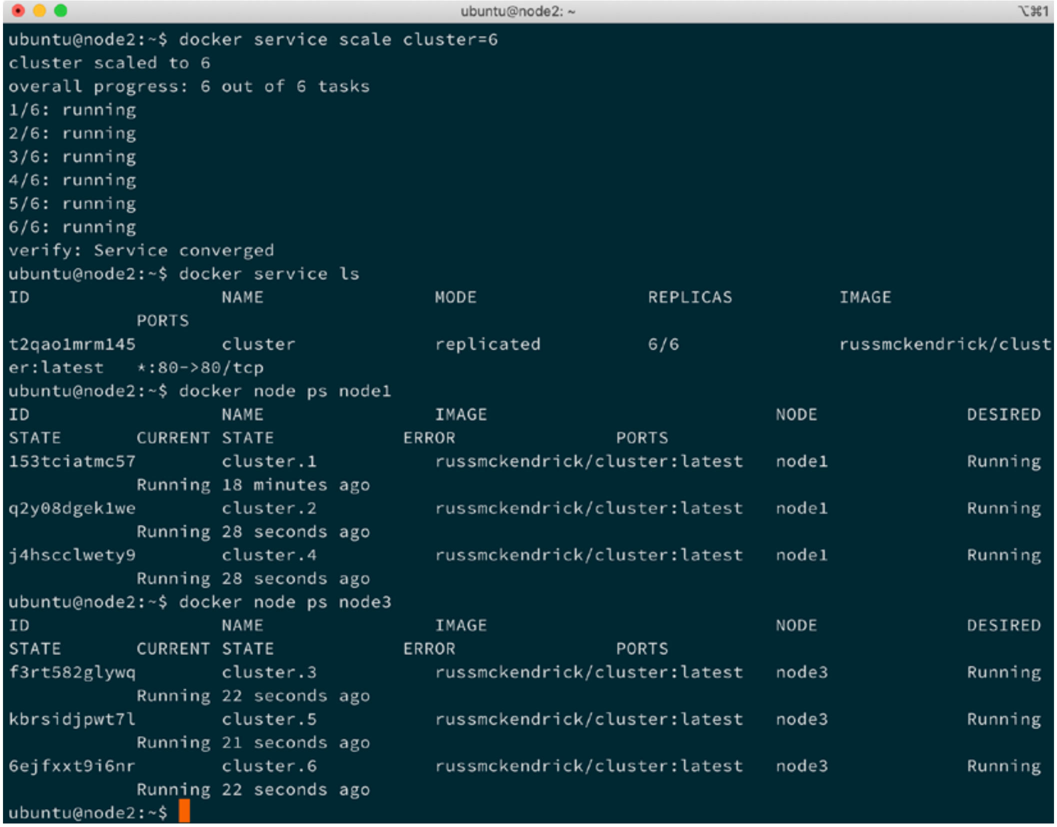
```
ubuntu@node2:~$ docker node ps
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE       ERROR               PORTS
ubuntu@node2:~$ docker node ps node1
ID                  NAME                IMAGE                           NODE                DESIRED
STATE        CURRENT STATE           ERROR               PORTS
153tciatmc57        cluster.1           russmckendrick/cluster:latest   node1               Running
            Running 13 minutes ago
ubuntu@node2:~$ docker node ps node3
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE       ERROR               PORTS
ubuntu@node2:~$
```

Figure 8.14 – Finding the node our service is running on

Let's look at scaling our service to six instances of our application container. Run the following commands to scale and check our service:

```
$ docker service scale cluster=6
$ docker service ls
$ docker node ps node1
$ docker node ps node3
```

We are only checking two of the nodes since we originally told our service to launch on worker nodes. As you can see from the following Terminal output, we now have three containers running on each of our worker nodes:



```
ubuntu@node2:~$ docker service scale cluster=6
cluster scaled to 6
overall progress: 6 out of 6 tasks
1/6: running
2/6: running
3/6: running
4/6: running
5/6: running
6/6: running
verify: Service converged
ubuntu@node2:~$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE
            PORTS
t2qao1mrm145        cluster             replicated          6/6                 russmckendrick/clust
er:latest   *:80->80/tcp
ubuntu@node2:~$ docker node ps node1
ID                  NAME                IMAGE                           NODE                DESIRED
STATE       CURRENT STATE            ERROR               PORTS
153tciatmc57        cluster.1           russmckendrick/cluster:latest   node1               Running
            Running 18 minutes ago
q2y08dgek1we        cluster.2           russmckendrick/cluster:latest   node1               Running
            Running 28 seconds ago
j4hscclwety9        cluster.4           russmckendrick/cluster:latest   node1               Running
            Running 28 seconds ago
ubuntu@node2:~$ docker node ps node3
ID                  NAME                IMAGE                           NODE                DESIRED
STATE       CURRENT STATE            ERROR               PORTS
f3rt582glywq        cluster.3           russmckendrick/cluster:latest   node3               Running
            Running 22 seconds ago
kbrsidjpwt7l        cluster.5           russmckendrick/cluster:latest   node3               Running
            Running 21 seconds ago
6ejfxxt9i6nr        cluster.6           russmckendrick/cluster:latest   node3               Running
            Running 22 seconds ago
ubuntu@node2:~$
```

Figure 8.15 – Checking the distribution of containers on our nodes

Before we move on to look at stacks, let's remove our service. To do this, run the following command:

```
$ docker service rm cluster
```

This will remove all of the containers, while leaving the downloaded image on the hosts.

Stacks

It is more than possible to create quite complex, highly available multi-container applications using Swarm services. In a non-Swarm cluster, manually launching each set of containers for a part of the application can start to become a little laborious and also makes it difficult to share services. To this end, Docker has created a functionality that allows you to define your services in Docker Compose files.

The following Docker Compose file, which is named `docker-compose.yml`, will create the same service we launched in the `services` section:

```
version: '3'

services:
  cluster:
    image: russmckendrick/cluster
    ports:
      - '80:80'
    deploy:
      replicas: 6
      restart_policy:
        condition: on-failure
      placement:
        constraints:
          - node.role == worker
```

As you can see, the stack can be made up of multiple services, each defined under the `services` section of the Docker Compose file.

In addition to the normal Docker Compose commands, you can add a `deploy` section; this is where you define everything relating to the Swarm element of your stack.

In the previous example, we said we would like six replicas, which should be distributed across our two worker nodes. Also, we updated the default restart policy, which you saw when we inspected the service from the previous section, and it showed up as `paused`, so that if a container becomes unresponsive, it is always restarted.

To launch our stack, copy the previous content into the `docker-compose.yml` file, and then run the following command:

```
$ docker stack deploy --compose-file=docker-compose.yml cluster
```

Docker will—as when launching containers with Docker Compose—create a new network and then launch your services on it. You can check the status of your stack by running this command:

```
$ docker stack ls
```

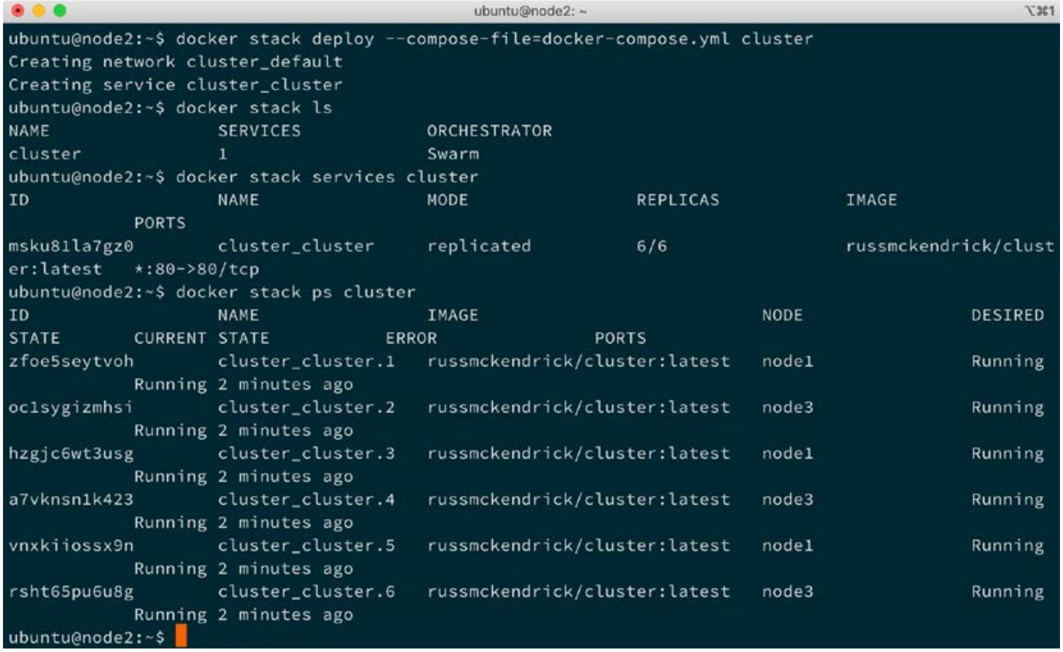
This will show that a single service has been created. You can get details of the service created by the `stack` by running this command:

```
$ docker stack services cluster
```

Finally, running the following command will show where the containers within the stack are running:

```
$ docker stack ps cluster
```

Take a look at the following Terminal output:



```
ubuntu@node2:~$ docker stack deploy --compose-file=docker-compose.yml cluster
Creating network cluster_default
Creating service cluster_cluster
ubuntu@node2:~$ docker stack ls
NAME                SERVICES            ORCHESTRATOR
cluster             1                   Swarm
ubuntu@node2:~$ docker stack services cluster
ID                  NAME                MODE                REPLICAS            IMAGE
            PORTS
msku81la7gz0        cluster_cluster     replicated          6/6                 russmckendrick/clust
er:latest   *:80->80/tcp
ubuntu@node2:~$ docker stack ps cluster
ID                  NAME                IMAGE                           NODE                DESIRED
STATE       CURRENT STATE           ERROR               PORTS
zfoe5seytvoh        cluster_cluster.1   russmckendrick/cluster:latest   node1               Running
            Running 2 minutes ago
oc1sygizmhsi        cluster_cluster.2   russmckendrick/cluster:latest   node3               Running
            Running 2 minutes ago
hzgjc6wt3usg        cluster_cluster.3   russmckendrick/cluster:latest   node1               Running
            Running 2 minutes ago
a7vknsn1k423        cluster_cluster.4   russmckendrick/cluster:latest   node3               Running
            Running 2 minutes ago
vnxkiiossx9n        cluster_cluster.5   russmckendrick/cluster:latest   node1               Running
            Running 2 minutes ago
rsht65pu6u8g        cluster_cluster.6   russmckendrick/cluster:latest   node3               Running
            Running 2 minutes ago
ubuntu@node2:~$
```

Figure 8.16 – Deploying our stack

Again, you will be able to access the stack using the IP addresses of your nodes, and you will be routed to one of the running containers. To remove a stack, simply run this command:

```
$ docker stack rm cluster
```

This will remove all services and networks created by the stack when it is launched.

Deleting a Swarm cluster

Before moving on, as we no longer require it for the next section, you can delete your Swarm cluster by running the following command:

```
$ multipass delete --purge node1
$ multipass delete --purge node2
$ multipass delete --purge node3
```

Should you need to relaunch the Swarm cluster for any reason, simply follow the instructions from the start of the chapter to recreate a cluster.

Load balancing, overlays, and scheduling

In the last few sections, we looked at launching services and stacks. To access the applications we launched, we were able to use any of the host IP addresses in our cluster; how was this possible?

Ingress load balancing

Docker Swarm has an ingress load balancer built in, making it easy to distribute traffic to our public-facing containers.

This means that you can expose applications within your Swarm cluster to services—for example, an external load balancer such as Amazon **Elastic Load Balancer** (**ELB**)—knowing that your request will be routed to the correct container(s) no matter which host happens to be currently hosting it, as demonstrated by the following diagram:

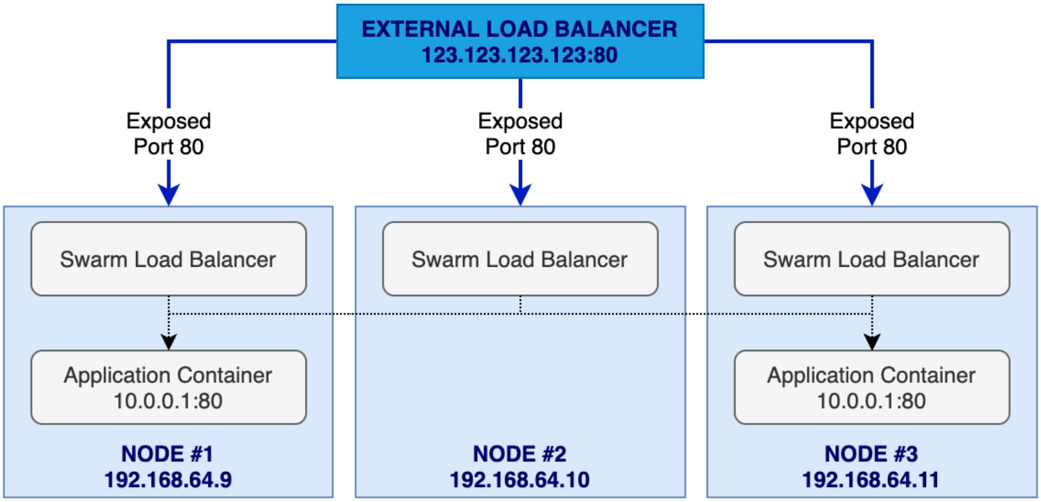


Figure 8.17 – An overview of load balancing in a Swarm cluster

This means that our application can be scaled up or down, fail, or be updated, all without the need to have the external load balancer reconfigured to talk to the individual containers, as Docker Swarm is handling that for us.

Network overlays

In our example, we launched a simple service running a single application. Say we wanted to add a database layer in our application, which is typically a fixed point within the network; how could we do this?

Docker Swarm's network overlay layer extends the network you launch your containers in across multiple hosts, meaning that each service or stack can be launched in its own isolated network. This means that our database container, running MongoDB, will be accessible to all other containers running on the same overlay network on port `27017`, no matter which of the hosts the containers are running on.

You may be thinking to yourself: *Hang on a minute. Does this mean I have to hardcode an IP address into my application's configuration?* Well, that wouldn't fit well with the problems Docker Swarm is trying to resolve, so no, you don't.

Each overlay network has its own inbuilt **DNS** (**Domain Name System**) service, which means that every container launched within the network is able to resolve the hostname of another container within the same network to its currently assigned IP address. This means that when we configure our application to connect to our database instance, we simply need to tell it to connect to, say, `mongodb:27017`, and it will connect to our MongoDB container.

This will make our diagram appear as follows:

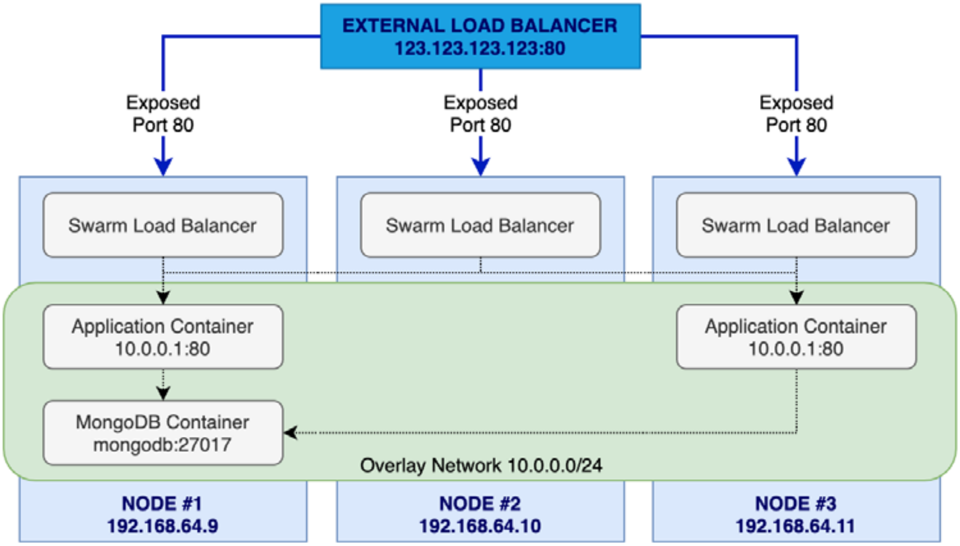


Figure 8.18 – An overview of overlay networks in a Docker Swarm cluster

There are some other considerations you will need to take into account when adopting this pattern, but we will cover those in *Chapter 15, Docker Workflows*.

Scheduling

At the time of writing, there is only a single scheduling strategy available within Docker Swarm, called **spread**. What this strategy does is schedule tasks to be run against the least loaded node that meets any of the constraints you defined when launching the service or stack. For the most part, you should not need to add too many constraints to your services.

One feature that is not currently supported by Docker Swarm is affinity and anti-affinity rules. While it is easy to get around using this constraint, I urge you not to overcomplicate things, as it is very easy to end up overloading hosts or creating single points of failure if you put too many constraints in place when defining your services.

Summary

In this chapter, we explored Docker Swarm. We took a look at how to install Docker Swarm and the Docker Swarm components that make up Docker Swarm. We took a look at how to use Docker Swarm, joining, listing, and managing Swarm manager and worker nodes. We reviewed the `service` and `stack` commands and how to use them, and spoke about the Swarm inbuilt ingress load balancer, overlay networks, and scheduler.

Now, Docker Swarm is in an interesting state at the time of writing, as Docker Swarm was one of the technologies acquired by Mirantis as part of the Docker Enterprise sale, and while Mirantis have said that they will offer support for existing Docker Swarm clusters for 2 years (that was in November 2019), they haven't given much information on the future of Docker Swarm.

This isn't surprising as Mirantis do a lot of work with another container cluster called Kubernetes, which we are going to be looking at in *Chapter 11*, *Docker and Kubernetes*. Before then, in the next chapter, we are going to take a look at a **graphical user interface** (**GUI**) for Docker called **Portainer**.

Questions

1. True or false: You should be running your Docker Swarm using the standalone Docker Swarm rather than the in-built Docker Swarm mode.
2. Which two things do you need after initiating your Docker Swarm manager to add your workers to your Docker Swarm cluster?
3. Which command would you use to find out the status of each of the nodes within your Docker Swarm cluster?
4. Which flag would you add to `docker node inspect` on Swarm manager to make it more readable?
5. How do you promote a node to be a manager?
6. Which command can you use to scale your service?

Further reading

For a detailed explanation of the Raft consensus algorithm, I recommend working through the excellent presentation entitled *The Secret Lives of Data*, which can be found at `http://thesecretlivesofdata.com/raft/`. It explains all the processes taking place in the background on the manager nodes via an easy-to-follow animation.

9
Portainer – A GUI for Docker

In this chapter, we will take a look at Portainer. Portainer is a tool that allows you to manage Docker resources from a web interface.

As Portainer itself is distributed in containers, it is simple to install and you can run it anywhere you can launch a container, making it the perfect interface for those who would prefer to not manage their containers using the command line as we have been doing in the previous chapters.

In this chapter, we will be covering the following topics:

- The road to Portainer
- Getting Portainer up and running
- Using Portainer
- Portainer and Docker Swarm

Technical requirements

As in previous chapters, we will continue to use our local Docker installations. Also, the screenshots in this chapter will be from my preferred operating system, macOS. Toward the end of the chapter, we will use Multipass to launch a local Docker Swarm cluster.

As before, the Docker commands we will be running will work on all three of the operating systems we have installed Docker on so far, however, some of the supporting commands, which will be few and far between, may only apply to macOS and Linux-based operating systems.

Check out the following video to see the Code in Action:
`https://bit.ly/3jR2HBa`

The road to Portainer

Before we roll up our sleeves and dive into installing and using Portainer, we should discuss the background of the project. The first edition of this book covered *Docker UI*. *Docker UI* was written by *Michael Crosby*, who handed the project over to *Kevan Ahlquist* after about a year of development. It was at this stage, due to trademark concerns, that the project was renamed UI for Docker.

The development of UI for Docker continued up until the point Docker started to accelerate the introduction of features such as Swarm mode into the core Docker Engine. It was around this time that the UI for Docker project was forked into the project that would become Portainer, which had its first major release in June 2016. Since their first public release, the team behind Portainer estimate the majority of the code has already been updated or rewritten, and by mid-2017, new features were added, such as role-based controls and Docker Compose support.

In December 2016, a notice was committed to the UI for Docker GitHub repository stating that the project was deprecated and that Portainer should be used. Since its release, it has been downloaded over 1.3 billion times.

Now we know a little about the background of Portainer, let's look at the steps needed to get it launched and configured.

Getting Portainer up and running

We are first going to be looking at using Portainer to manage a single Docker instance running locally. I am running Docker for Mac so I will be using that, but these instructions should also work with other Docker installations.

First of all, to grab the container image from Docker Hub, we just need to run the following commands:

```
$ docker image pull portainer/portainer
$ docker image ls
```

As you may see from the output if you are following along when we run the `docker image ls` command, the Portainer image is only 78.6 MB. To launch Portainer, you simply have to run the following command if you are running macOS or Linux:

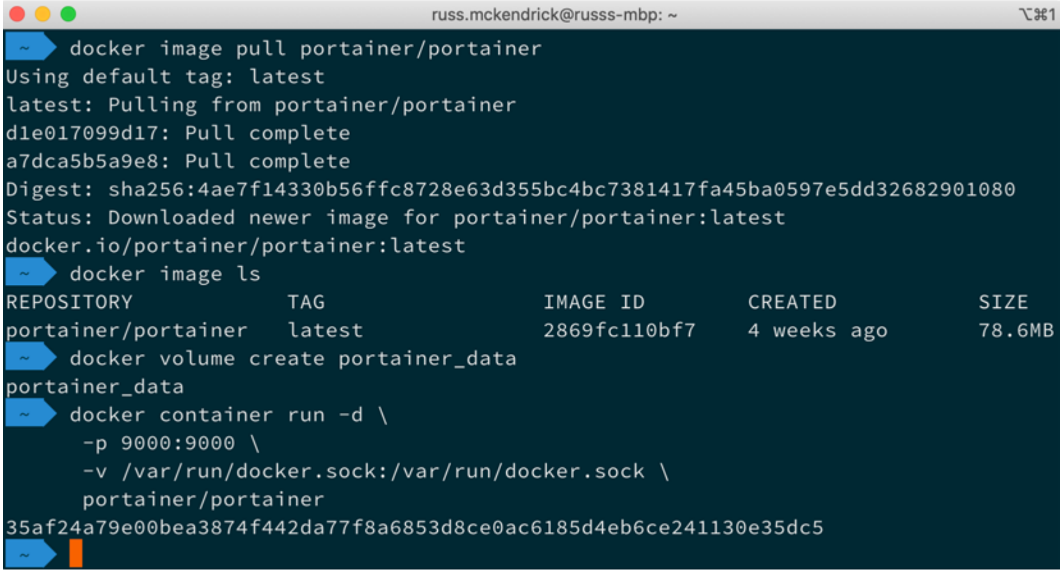
```
$ docker volume create portainer_data
$ docker container run -d \
      -p 9000:9000 \
      -v /var/run/docker.sock:/var/run/docker.sock \
      portainer/portainer
```

Windows users will have to run the following:

```
$ docker container run -d -p 9000:9000 -v \\.\pipe\docker_
engine:\\.\pipe\docker_engine portainer/portainer
```

As you can see from the command we have just run, we are mounting the socket file for Docker Engine on our Docker host machine. Doing this will allow Portainer full, unrestricted access to Docker Engine on our host machine. It needs this so it can manage Docker on the host; however, it does mean that your Portainer container has full access to your host machine, so be careful in how you give access to it and also when publicly exposing Portainer on remote hosts.

The following screenshot shows this being executed on macOS:



```
~  docker image pull portainer/portainer
Using default tag: latest
latest: Pulling from portainer/portainer
d1e017099d17: Pull complete
a7dca5b5a9e8: Pull complete
Digest: sha256:4ae7f14330b56ffc8728e63d355bc4bc7381417fa45ba0597e5dd32682901080
Status: Downloaded newer image for portainer/portainer:latest
docker.io/portainer/portainer:latest
~  docker image ls
REPOSITORY           TAG                 IMAGE ID       CREATED          SIZE
portainer/portainer   latest              2869fc110bf7   4 weeks ago      78.6MB
~  docker volume create portainer_data
portainer_data
~  docker container run -d \
     -p 9000:9000 \
     -v /var/run/docker.sock:/var/run/docker.sock \
     portainer/portainer
35af24a79e00bea3874f442da77f8a6853d8ce0ac6185d4eb6ce241130e35dc5
~  
```

Figure 9.1 – Downloading and launching the Portainer container

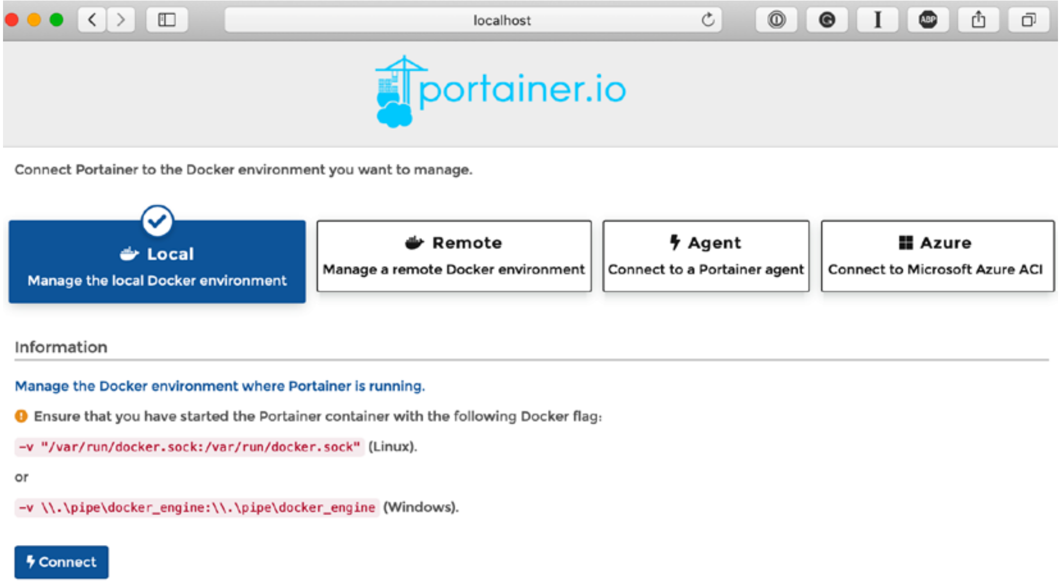
For the most basic type of installation, that is all we need to run. There are a few more steps to complete the installation; they are all performed in the browser. To complete them, go to `http://localhost:9000/`. The first screen you will be greeted by asks you to set a password for the admin user.

Once you have set the password, you will be taken to a login page: enter the username admin and the password you just configured. Once logged in, you will be asked about the Docker instance you wish to manage.

There are four options:

- Manage the Docker instance where Portainer is running
- Manage a remote Docker instance
- Connect to a Portainer Agent
- Connect to Microsoft **Azure Container Instances** (**ACI**)

For the moment, we want to manage the instance where Portainer is running, which is the **Local** option, rather than the default **Remote** one:



Connect Portainer to the Docker environment you want to manage.

Local
Manage the local Docker environment
Remote
Manage a remote Docker environment
Agent
Connect to a Portainer agent
Azure
Connect to Microsoft Azure ACI

Information

Manage the Docker environment where Portainer is running.

Ensure that you have started the Portainer container with the following Docker flag:

`-v "/var/run/docker.sock:/var/run/docker.sock"` (Linux).

or

`-v \\.\pipe\docker_engine:\\.\pipe\docker_engine` (Windows).

Figure 9.2 – Choosing which environment you want to manage with Portainer

As we have already taken mounting the Docker socket file into account when launching our Portainer container, we can click on **Connect** to complete our installation. This will take us straight into Portainer itself, showing us the dashboard. With Portainer launched and configured, we can now look at some of the features.

Using Portainer

Now that we have Portainer running and configured to communicate with our Docker installation, we can start to work through the features listed in the left-hand side menu, starting at the top with the Dashboard, which is also the default landing page of your Portainer installation, as you can see from the following screenshot:

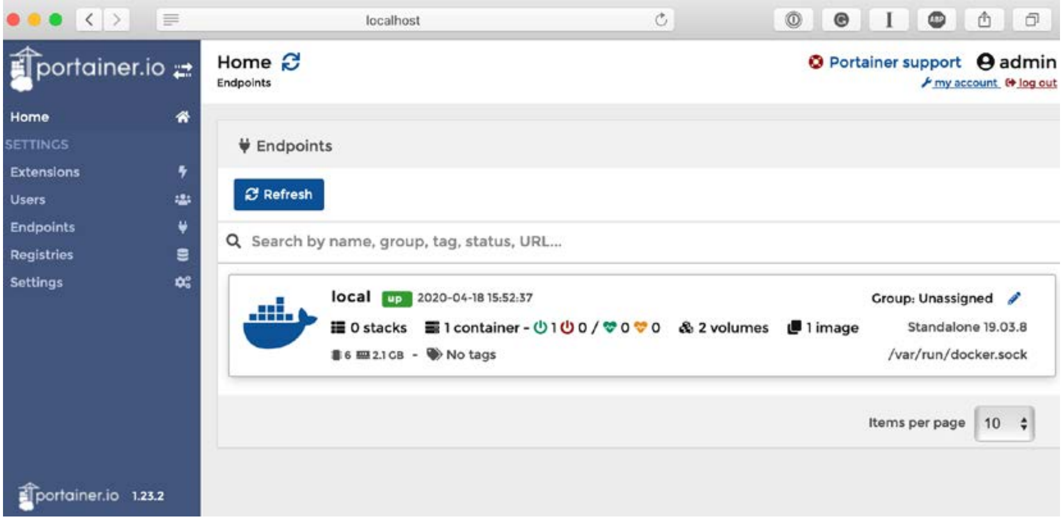
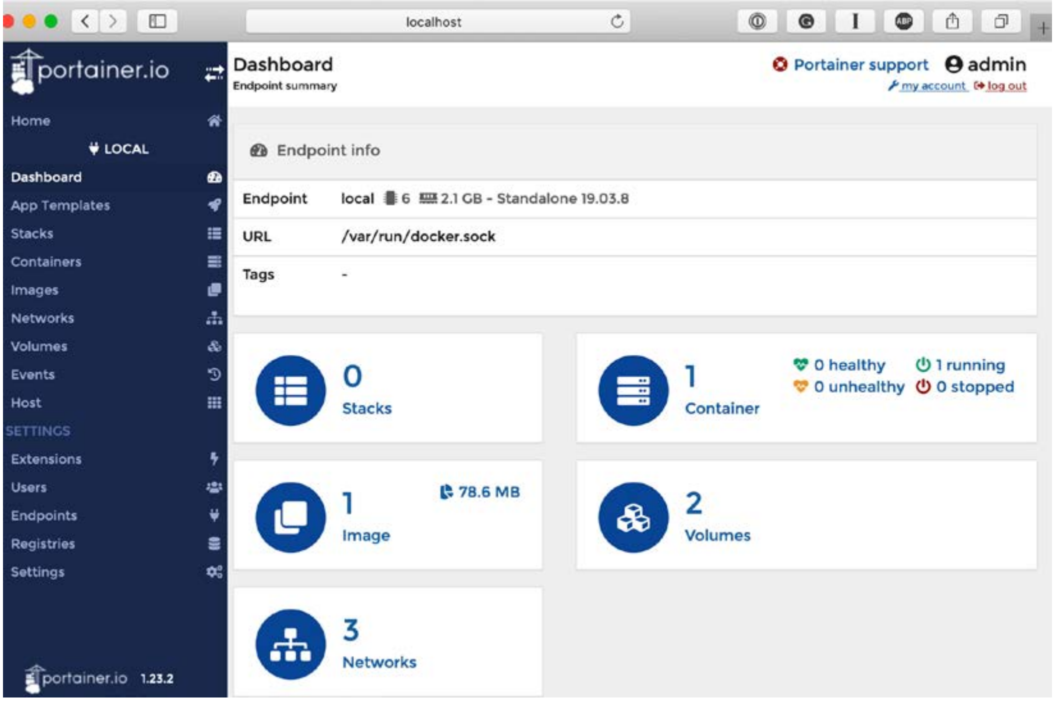


Figure 9.3 – Viewing the default page

You are first taken to the list of endpoints. As we only have our local installation, click on that and then we can start exploring.

The dashboard

As you can see from the following screenshot, the dashboard gives us an overview of the current state of the Docker instance that Portainer is configured to communicate with:



Dashboard
Endpoint summary

Endpoint info

Endpoint	local 6 2.1 GB - Standalone 19.03.8
URL	/var/run/docker.sock
Tags	-

0 Stacks

1 Container — 0 healthy, 0 unhealthy, 1 running, 0 stopped

1 Image — 78.6 MB

2 Volumes

3 Networks

Figure 9.4 – Getting an overview

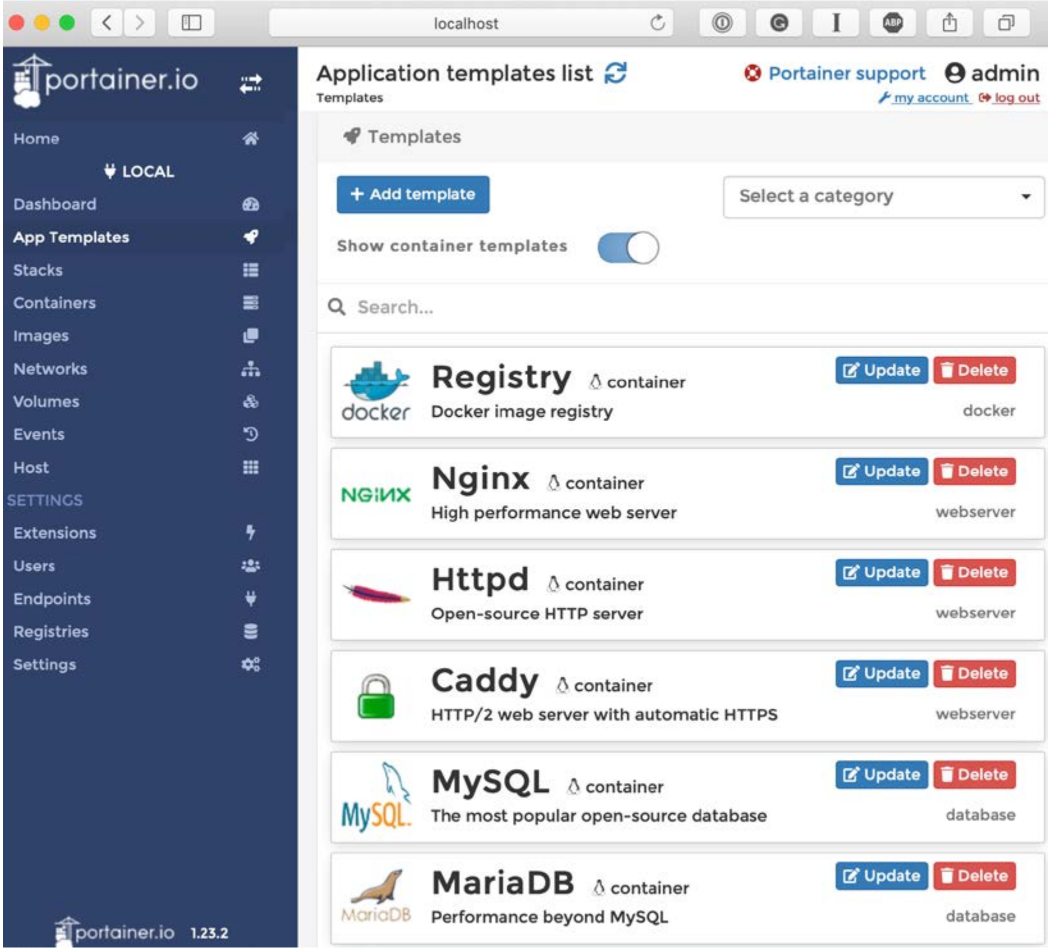
In my case, this shows how many containers I have running, which at the moment is just the already running Portainer container, as well as the number of images I have downloaded. We can also see the number of **Volumes** and **Networks** available on the Docker instance. It will also show the number of running **Stacks**:

It also shows basic information on the Docker instance itself; as you can see, the Docker instance is running Moby Linux, and has 6 CPUs and 2.1 GB of RAM. This is the default configuration for Docker for Mac.

The dashboard will adapt to the environment you have Portainer running in, so we will revisit it when we look at attaching Portainer to a Docker Swarm cluster.

Application templates

Next up in the left-hand menu, we have **App Templates**. This section is probably the only feature not to be a direct feature available in the core Docker Engine; it is instead a way of launching common applications using containers downloaded from Docker Hub:



Registry container Docker image registry Update Delete docker

Nginx container High performance web server Update Delete webserver

Httpd container Open-source HTTP server Update Delete webserver

Caddy container HTTP/2 web server with automatic HTTPS Update Delete webserver

MySQL container The most popular open-source database Update Delete database

MariaDB container Performance beyond MySQL Update Delete database

Figure 9.5 – Exploring the templates

There are around 25 templates that ship with Portainer by default. The templates are defined in JSON format. For example, the NGINX template looks like the following:

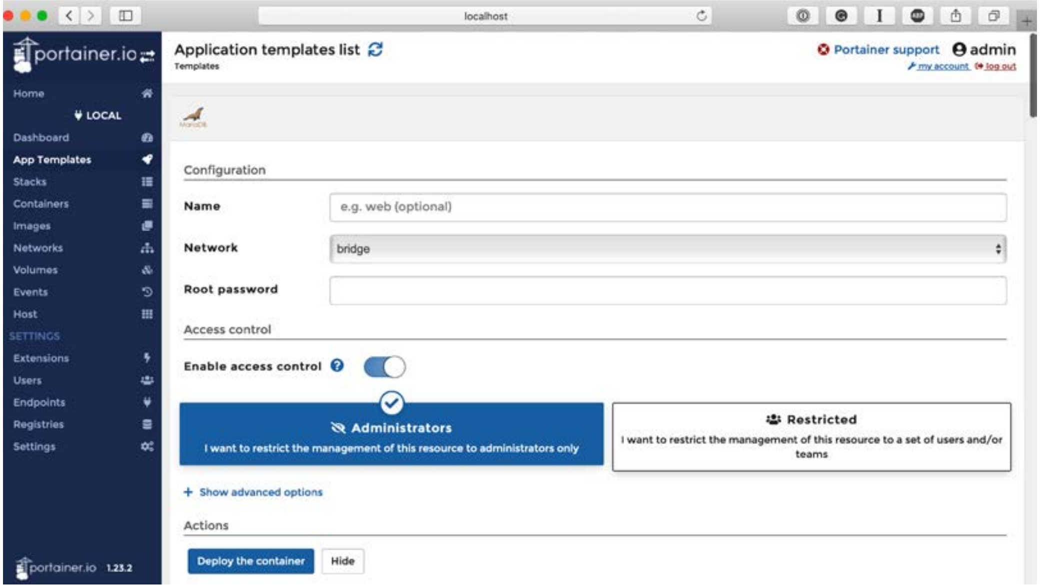
```
{
   'type': 'container',
   'title': 'Nginx',
   'description': 'High performance web server',
   'categories': ['webserver'],
   'platform': 'linux',
   'logo': 'https://portainer.io/images/logos/nginx.png',
   'image': 'nginx:latest',
   'ports': [
     '80/tcp',
     '443/tcp'
   ],
   'volumes': ['/etc/nginx', '/usr/share/nginx/html']
}
```

There are more options you can add, for example, the MariaDB template:

```
  {
    'type': 'container',
    'title': 'MariaDB',
    'description': 'Performance beyond MySQL',
    'categories': ['database'],
    'platform': 'linux',
    'logo': 'https://portainer.io/images/logos/mariadb.png',
    'image': 'mariadb:latest',
    'env': [
      {
        'name': 'MYSQL_ROOT_PASSWORD',
        'label': 'Root password'
      } ],
    'ports': ['3306/tcp' ],
    'volumes': ['/var/lib/mysql']
  }
```

As you can see, the templates look similar to a Docker Compose file; however, this format is only used by Portainer. For the most part, the options are pretty self-explanatory, but we should touch upon the **Name** and **Label** options.

For containers that typically require options defined by passing custom values via environment variables, the **Name** and **Label** options allow you to present the user with custom form fields that need to be completed before the container is launched, as demonstrated by the following screenshot:



Configuration

Name: e.g. web (optional)

Network: bridge

Root password:

Access control

Enable access control: on

☑ Administrators – I want to restrict the management of this resource to administrators only

Restricted – I want to restrict the management of this resource to a set of users and/or teams

Figure 9.6 – Launching MariaDB using the template

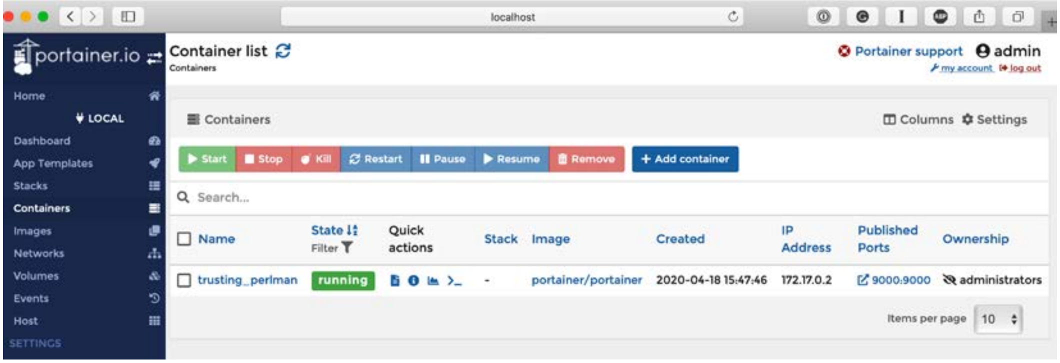
As you can see, we have a field where we can enter the root password we would like to use for our MariaDB container. Filling this in will take that value and pass it as an environment variable, building the following command to launch the container:

```
$ docker container run --name [Name of Container] -p 3306 -e
MYSQL_ROOT_PASSWORD=[Root password] -d mariadb:latest
```

For more information on app templates, I recommend reviewing the documentation – a link to this can be found in the *Further reading* section of this chapter.

Containers

The next thing we are going to look at in the left-hand menu is **Containers**. This is where you launch and interact with the containers running on your Docker instance. Clicking on the **Containers** menu entry will bring up a list of all of the containers, both running and stopped, on your Docker instance:



☐ Name	State Filter	Quick actions	Stack	Image	Created	IP Address	Published Ports	Ownership
☐ trusting_perlman	running		-	portainer/portainer	2020-04-18 15:47:46	172.17.0.2	9000:9000	administrators

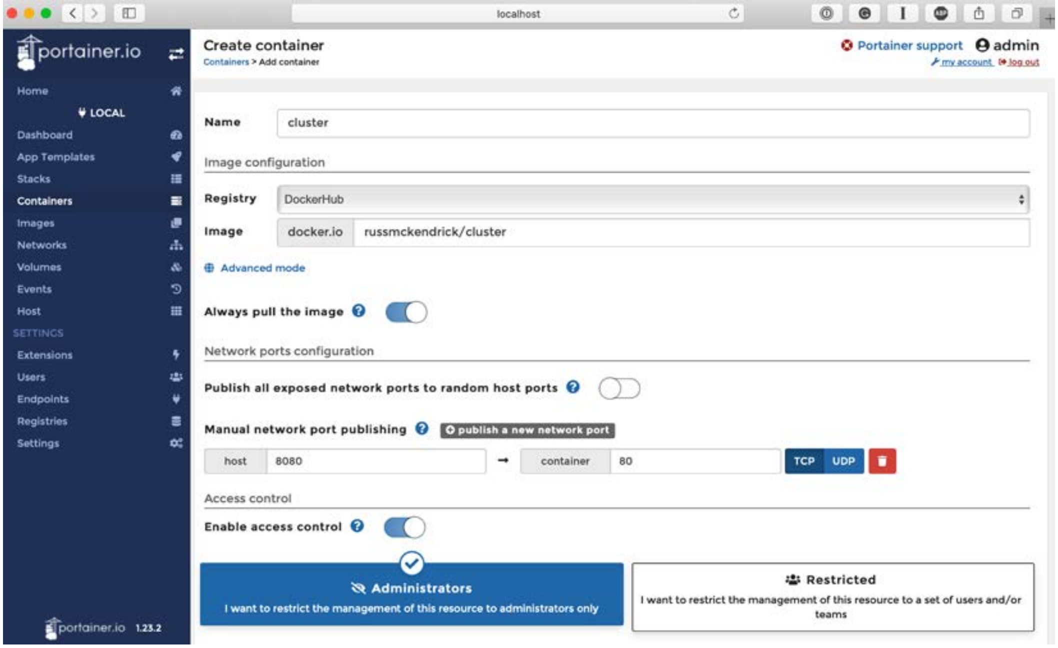
Figure 9.7 – Listing the containers

As you can see, I currently have only a single container running, and that just happens to be the Portainer one. Rather than interacting with that, let's click the **+ Add container** button to launch a container running the cluster application we used in previous chapters.

There are several options on the **Create container** page; these should be filled in as follows:

- **Name**: `cluster`
- **Image**: `russmckendrick/cluster`
- **Always pull the image**: On
- **Publish all exposed network ports to random host ports**: On

Finally, add a port mapping from port 8080 on the host to port 80 on the container by clicking on **+ publish a new network port**. Your completed form should look something like the following screenshot:



Create container

Containers > Add container

Name: cluster

Image configuration

Registry: DockerHub

Image: docker.io russmckendrick/cluster

Always pull the image: on

Network ports configuration

Publish all exposed network ports to random host ports: off

Manual network port publishing

host 8080 → container 80

Access control

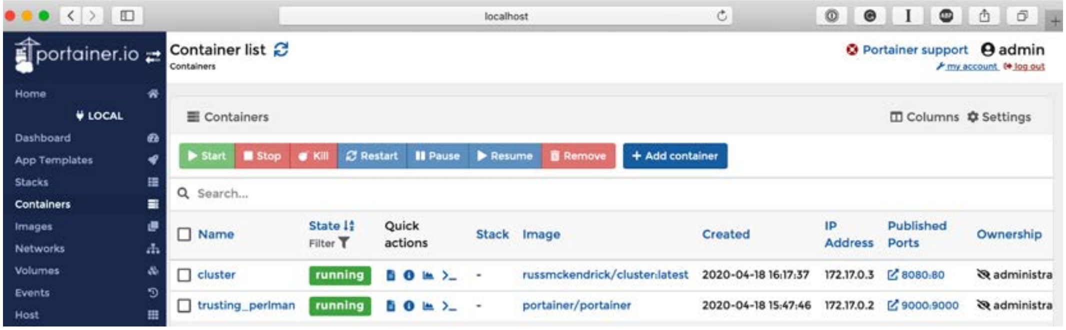
Enable access control: on

Administrators – I want to restrict the management of this resource to administrators only

Restricted – I want to restrict the management of this resource to a set of users and/or teams

Figure 9.8 – Launching a container

Once that's done, click on **Deploy the container**, and after a few seconds, the list of running containers will be returned, where you should see your newly launched container:



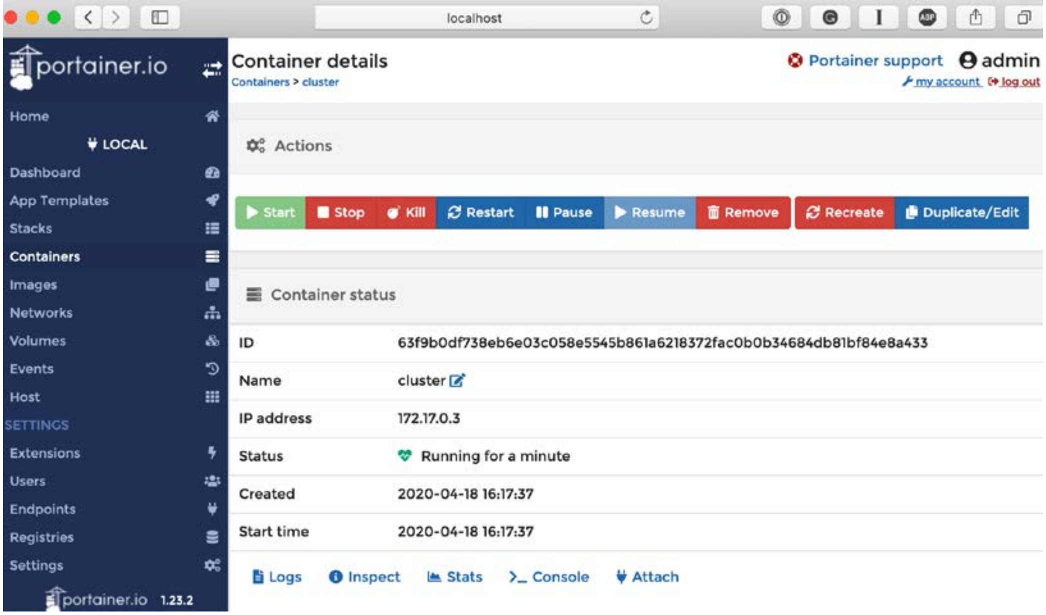
Container list

Containers

Name	State Filter	Quick actions	Stack	Image	Created	IP Address	Published Ports	Ownership
cluster	running		-	russmckendrick/cluster:latest	2020-04-18 16:17:37	172.17.0.3	8080:80	administra
trusting_perlman	running		-	portainer/portainer	2020-04-18 15:47:46	172.17.0.2	9000:9000	administra

Figure 9.9 – Listing the containers

Using the tick box on the left of each container in the list will enable the buttons at the top, where you can control the status of your containers. Make sure not to **Kill** or **Remove** the Portainer container. Clicking on the name of the container in our case cluster will bring up more information on the container itself:



ID	63f9b0df738eb6e03c058e5545b861a6218372fac0b0b34684db81bf84e8a433
Name	cluster
IP address	172.17.0.3
Status	Running for a minute
Created	2020-04-18 16:17:37
Start time	2020-04-18 16:17:37

Figure 9.10 – Drilling down into our container

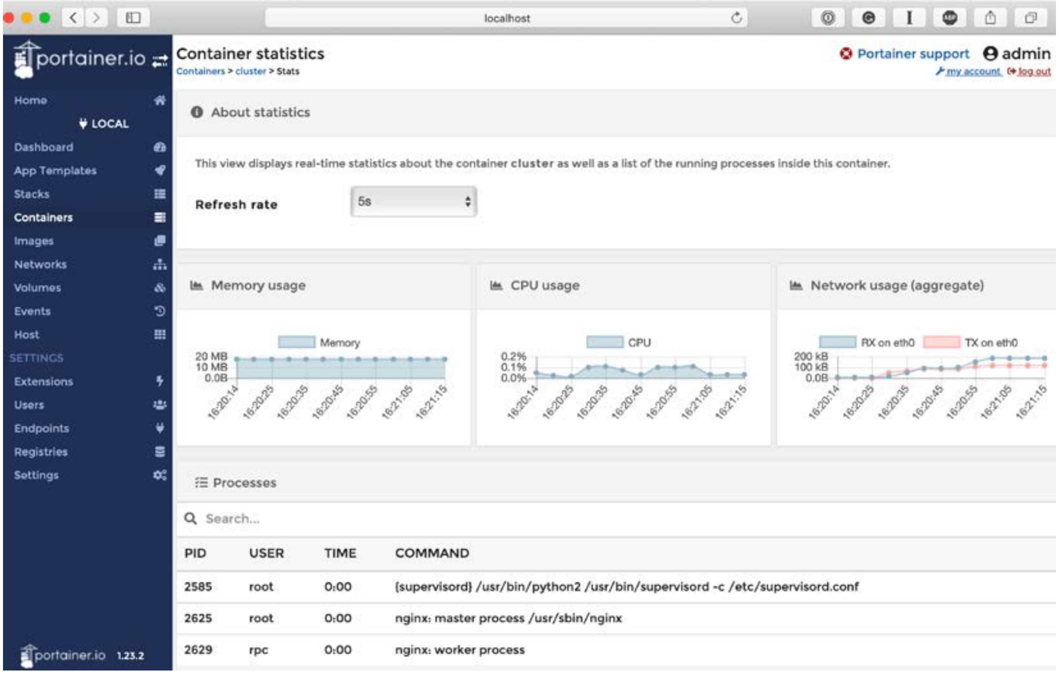
As you can see, the information about the container is the same information you would get if you were to run this command:

```
$ docker container inspect cluster
```

You can see the full output of this command by clicking on **Inspect**. You will also notice that there are buttons for **Stats**, **Logs**, **Console**, and **Attach**, which we will be discussing next.

Stats

The **Stats** page shows the CPU, memory, and network utilization, as well as a list of the processes for the container you are inspecting:



Container statistics

Containers > cluster > Stats

About statistics

This view displays real-time statistics about the container **cluster** as well as a list of the running processes inside this container.

Refresh rate: 5s

PID	USER	TIME	COMMAND
2585	root	0:00	(supervisord) /usr/bin/python2 /usr/bin/supervisord -c /etc/supervisord.conf
2625	root	0:00	nginx: master process /usr/sbin/nginx
2629	rpc	0:00	nginx: worker process

Figure 9.11 – Viewing the container stats

The graphs will automatically refresh if you leave the page open, and refreshing the page will zero the graphs and start afresh. This is because Portainer is receiving this information from the Docker API using the following command:

```
$ docker container stats cluster
```

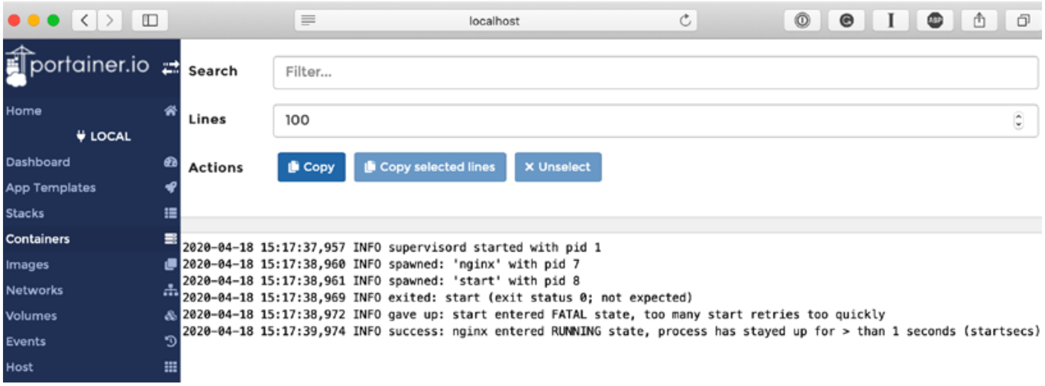
Each time the page is refreshed, the command is started from scratch as Portainer currently does not poll Docker in the background to keep a record of statistics for each of the running containers.

Logs

Next up, we have the **Logs** page. This shows you the results of running the following command:

```
$ docker container logs cluster
```

It displays both the STDOUT and STDERR logs:



Search: Filter...

Lines: 100

Actions

```
2020-04-18 15:17:37,957 INFO supervisord started with pid 1
2020-04-18 15:17:38,960 INFO spawned: 'nginx' with pid 7
2020-04-18 15:17:38,961 INFO spawned: 'start' with pid 8
2020-04-18 15:17:38,969 INFO exited: start (exit status 0; not expected)
2020-04-18 15:17:38,972 INFO gave up: start entered FATAL state, too many start retries too quickly
2020-04-18 15:17:39,974 INFO success: nginx entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
```

Figure 9.12 – Viewing the container logs

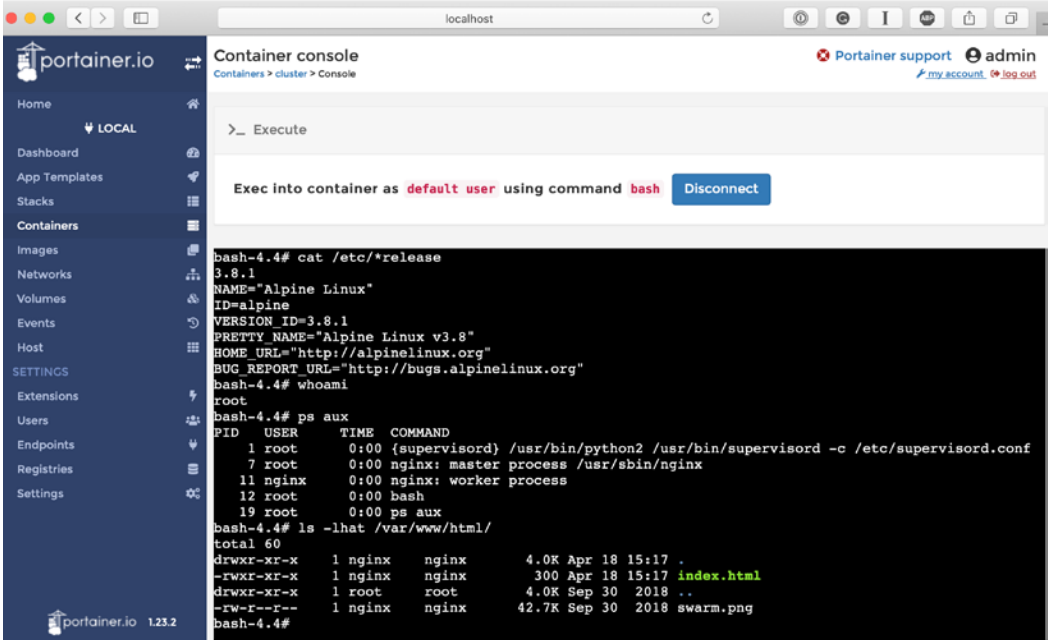
You also have the option of adding timestamps to the output; this is the equivalent of running the following:

```
$ docker container logs --timestamps cluster
```

As we have previously discussed, remember that the timestamps may be different depending on your host machine's time zone settings.

Console and Attach

Next up, we have **Console**. This will open an HTML5 terminal and allow you to log in to your running container. Before you connect to your container, you need to choose a shell. You have the option of three shells to use: /bin/bash , bin/sh, or /bin/ash, and also which user to connect as – root is the default. While the cluster image has both shells installed, I choose to use /bin/bash:



Exec into container as default user using command bash

```
bash-4.4# cat /etc/*release
3.8.1
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.8.1
PRETTY_NAME="Alpine Linux v3.8"
HOME_URL="http://alpinelinux.org"
BUG_REPORT_URL="http://bugs.alpinelinux.org"
bash-4.4# whoami
root
bash-4.4# ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 {supervisord} /usr/bin/python2 /usr/bin/supervisord -c /etc/supervisord.conf
    7 root      0:00 nginx: master process /usr/sbin/nginx
   11 nginx     0:00 nginx: worker process
   12 root      0:00 bash
   19 root      0:00 ps aux
bash-4.4# ls -lhat /var/www/html/
total 60
drwxr-xr-x    1 nginx    nginx       4.0K Apr 18 15:17 .
-rwxr-xr-x    1 nginx    nginx        300 Apr 18 15:17 index.html
drwxr-xr-x    1 root     root        4.0K Sep 30  2018 ..
-rw-r--r--    1 nginx    nginx      42.7K Sep 30  2018 swarm.png
bash-4.4#
```

Figure 9.13 – Opening a session to the container

This is the equivalent of running the following command to gain access to your container:

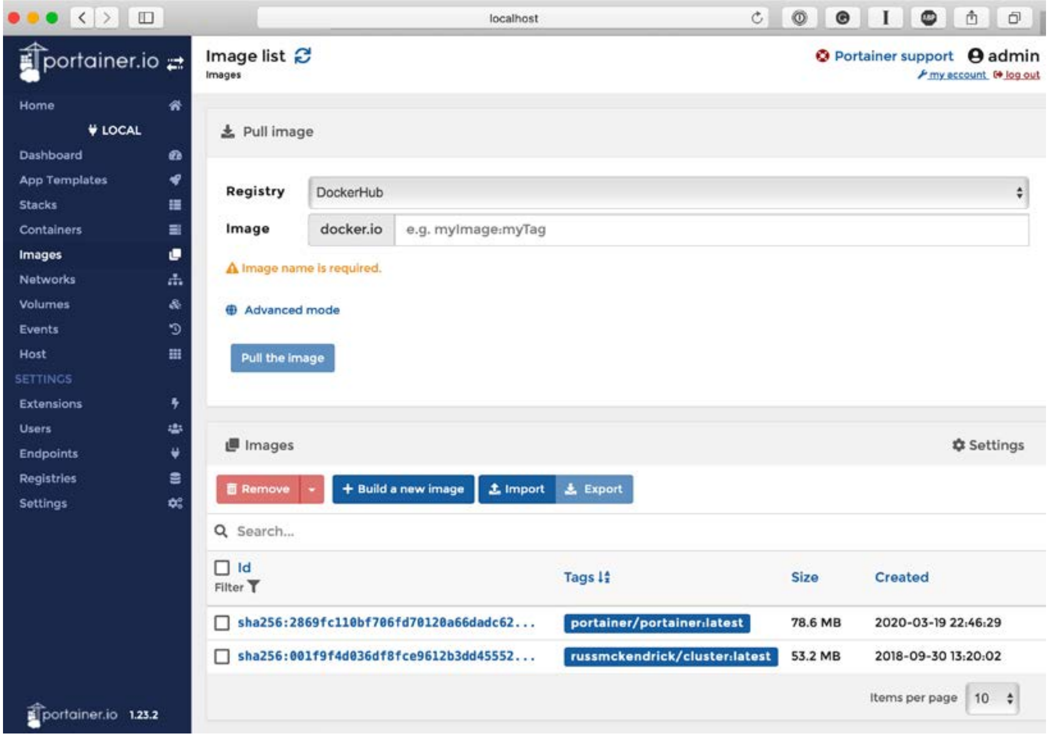
```
$ docker container exec -it cluster /bin/sh
```

As you can see from the screenshot, the bash process has a PID of 15. This process was created by the docker container exec command, and that will be the only process that is terminated once you disconnect from your shell session.

If we had launched our container with the `TTY` flag, we could have also used the **Attach** button to connect to the `TTY` of the container rather than spawning a shell to attach to, as we did when using **Console**, like the like when attaching on the command line your process will stop when disconnect.

Images

Next up in the left-hand menu is **Images**. From here, you can manage, download, and upload images:



Registry: DockerHub

Image: docker.io e.g. myImage:myTag

⚠ Image name is required.

□ Id Filter	Tags	Size	Created
□ sha256:2869fc110bf706fd70120a66dadc62...	portainer/portainer:latest	78.6 MB	2020-03-19 22:46:29
□ sha256:001f9f4d036df8fce9612b3dd45552...	russmckendrick/cluster:latest	53.2 MB	2018-09-30 13:20:02

Figure 9.14 – Managing your images

At the top of the page, you have the option of pulling an image. For example, simply entering `amazonlinux` into the box and then clicking on **Pull the image** will download a copy of the Amazon Linux container image from Docker Hub. The command executed by Portainer would be this:

```
$ docker image pull amazonlinux:latest
```

You can find more information about each image by clicking on the image ID; this will take you to a page that nicely renders the output of running this command:

```
$ docker image inspect russmckendrick/cluster
```

Look at the following screenshot:

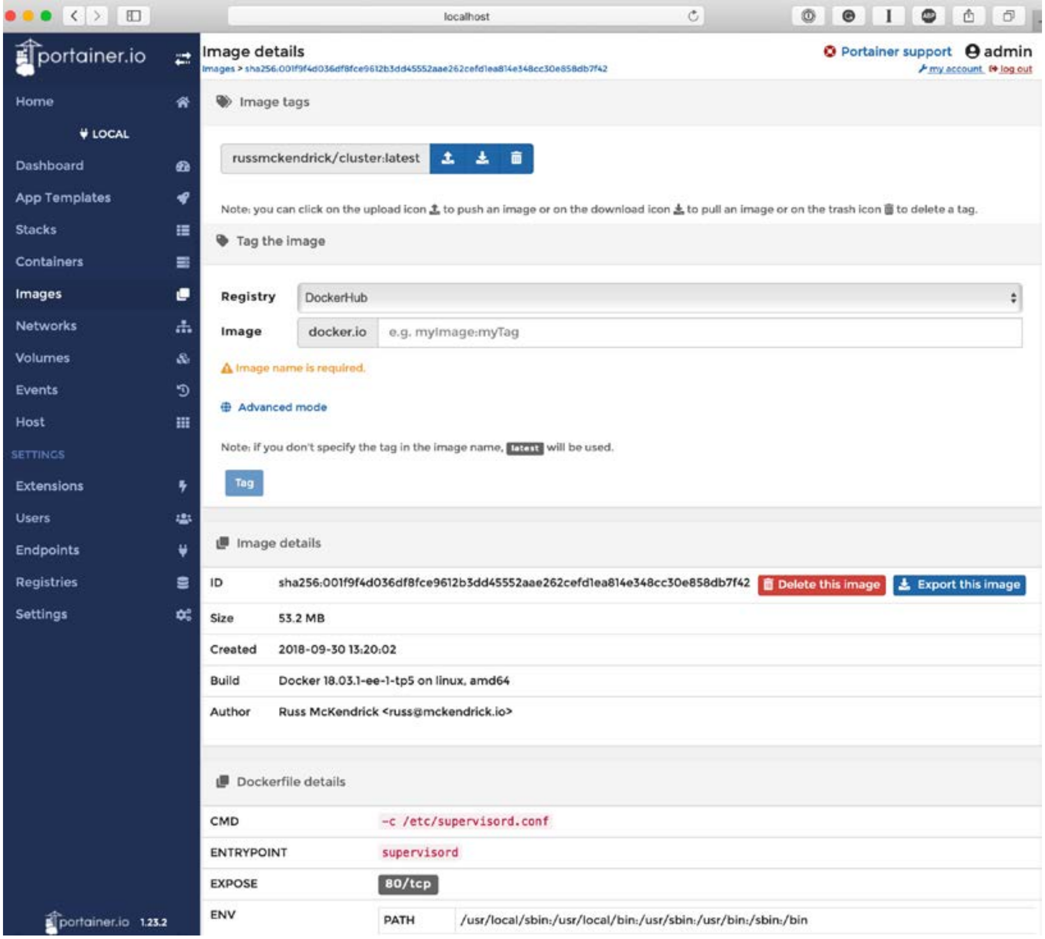


Image details

Images > sha256:001f9f4d036df8fce9612b3dd45552aae262cefd1ea814e348cc30e858db7f42

Image tags

russmckendrick/cluster:latest

Note: you can click on the upload icon to push an image or on the download icon to pull an image or on the trash icon to delete a tag.

Tag the image

Registry: DockerHub

Image: docker.io e.g. myImage:myTag

⚠ Image name is required.

Advanced mode

Note: if you don't specify the tag in the image name, latest will be used.

Tag

Image details

ID	sha256:001f9f4d036df8fce9612b3dd45552aae262cefd1ea814e348cc30e858db7f42 Delete this image Export this image
Size	53.2 MB
Created	2018-09-30 13:20:02
Build	Docker 18.03.1-ee-1-tp5 on linux, amd64
Author	Russ McKendrick <russ@mckendrick.io>

Dockerfile details

CMD	-c /etc/supervisord.conf
ENTRYPOINT	supervisord
EXPOSE	80/tcp
ENV	PATH /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

Figure 9.15 – Getting more information on your image

Not only do you get all of the information about the image, but you also get options to push a copy of the image to your chosen registry or, by default, Docker Hub. You also get a complete breakdown of each of the layers contained within the image, showing the command that was executed during the build and the size of each layer.

The next two items in the menu allow you to manage networks and volumes; I am not going to go into too much detail here as there is not much to them.

Networks

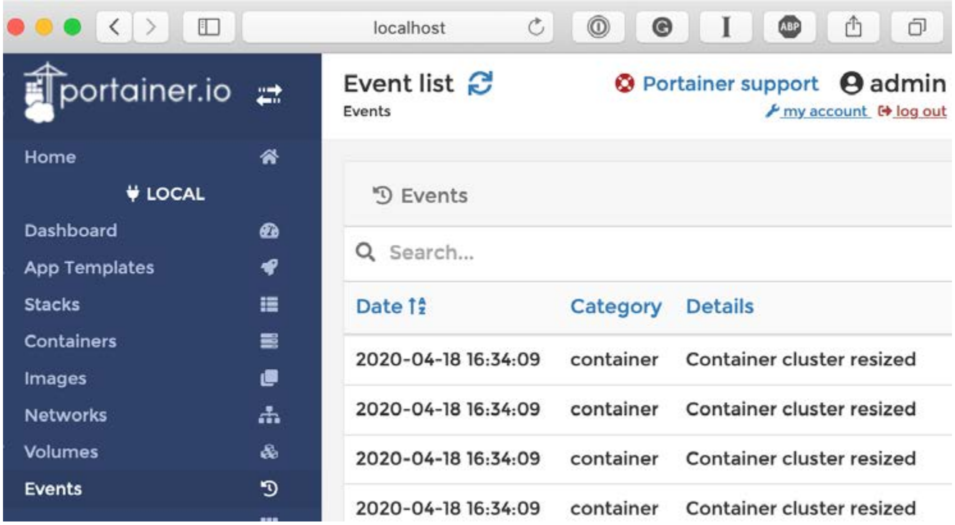
Here, you can quickly add a network using the default bridge driver. Clicking on **Advanced settings** will take you to a page with more options. These include using other drivers, defining the subnets, adding labels, and restricting external access to the network. As with other sections, you can also remove networks and inspect existing networks.

Volumes

There are not many options here other than adding or removing a volume. When adding a volume, you get a choice of drivers as well as being able to fill in options to pass to the driver, which allows the use of third-party driver plugins. Other than that, there is not much to see here, not even an inspect option.

Events

The events page shows you all of the events from the last 24 hours; you also have an option of filtering the results, meaning you can quickly find the information you are after:



Date	Category	Details
2020-04-18 16:34:09	container	Container cluster resized
2020-04-18 16:34:09	container	Container cluster resized
2020-04-18 16:34:09	container	Container cluster resized
2020-04-18 16:34:09	container	Container cluster resized

Figure 9.16 – Viewing Docker events

This is the equivalent of running the following command:

```
$ docker events --since '2020-04-17T16:30:00' --until '2020-04-
17T16:30:00'
```

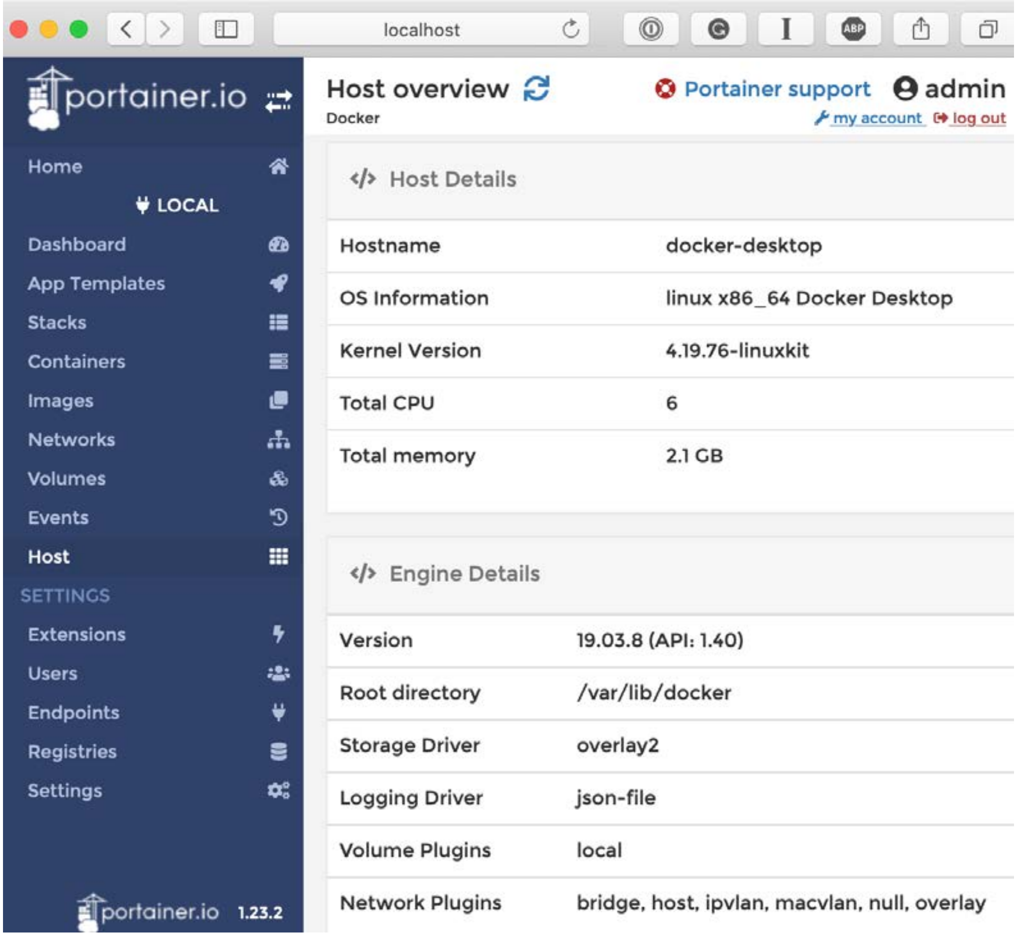
This leaves us with one more option to cover.

Host

The final entry simply shows you the output of the following:

```
$ docker info
```

The following shows the output of the command:



Host overview
Docker

Host Details

Hostname	docker-desktop
OS Information	linux x86_64 Docker Desktop
Kernel Version	4.19.76-linuxkit
Total CPU	6
Total memory	2.1 GB

Engine Details

Version	19.03.8 (API: 1.40)
Root directory	/var/lib/docker
Storage Driver	overlay2
Logging Driver	json-file
Volume Plugins	local
Network Plugins	bridge, host, ipvlan, macvlan, null, overlay

Figure 9.17 – Viewing information on the Docker host

This can be useful if you are targeting multiple Docker instance endpoints and need information on the environment the endpoint is running on.

At this point, we are moving on to looking at Portainer running on Docker Swarm, so now would be a good time to remove the running containers and also the volume that was created when we first launched Portainer. You can remove the volume using the following:

```
$ docker volume prune
```

Now we have cleaned up, let's look at launching a Docker Swarm cluster.

Portainer and Docker Swarm

In the previous section, we looked at how to use Portainer on a standalone Docker instance. Portainer also supports Docker Swarm clusters, and the options in the interface adapt to the clustered environment. We should look at spinning up a Swarm and then launching Portainer as a service and see what changes.

So let's start by launching a new Docker Swarm cluster.

Creating the Swarm

As in the Docker Swarm chapter, we are going to be creating the Swarm locally using Multipass; to do this, run the following commands to launch the three nodes:

```
$ multipass launch -n node1
$ multipass launch -n node2
$ multipass launch -n node3
```

Now install Docker:

```
$ multipass exec node1 -- \
   /bin/bash -c 'curl -s https://get.docker.com | sh - && sudo
usermod -aG docker ubuntu'

$ multipass exec node2 -- \
   /bin/bash -c 'curl -s https://get.docker.com | sh - && sudo
usermod -aG docker ubuntu'

$ multipass exec node3 -- \
   /bin/bash -c 'curl -s https://get.docker.com | sh - && sudo
usermod -aG docker ubuntu'
```

Once Docker is installed, initialize and create the cluster:`P=$(multipass info node1 | grep IPv4 | a`

```

$ multipass exec node1 -- \
   /bin/bash -c 'docker swarm init --advertise-addr $IP:2377
--listen-addr $IP:2377'

$ SWARM_TOKEN=$(multipass exec node1 -- /bin/bash -c 'docker
swarm join-token --quiet worker')

$ multipass exec node2 -- \
   /bin/bash -c 'docker swarm join --token $SWARM_TOKEN
$IP:2377'

$ multipass exec node3 -- \
   /bin/bash -c 'docker swarm join --token $SWARM_TOKEN
$IP:2377'
```

Make a note of the IP address of `node1` by running the following command:

```
$ multipass list
```

Finally, log in to the Swarm manager node:

```
$ multipass shell node1
```

You should now be logged into the primary Swarm node and ready to progress.

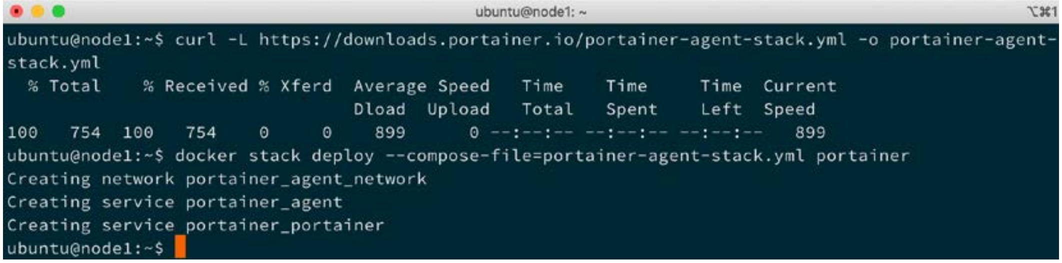
The Portainer service

Now that we have a Docker Swarm cluster, we can launch a Portainer stack by simply running the following:

```
$ curl -L https://downloads.portainer.io/portainer-agent-stack.
yml -o portainer-agent-stack.yml
```

```
$ docker stack deploy --compose-file=portainer-agent-stack.yml
portainer
```

This should give you something that looks like the following output:



```
ubuntu@node1:~$ curl -L https://downloads.portainer.io/portainer-agent-stack.yml -o portainer-agent-
stack.yml
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   754  100   754    0     0    899      0 --:--:-- --:--:-- --:--:--   899
ubuntu@node1:~$ docker stack deploy --compose-file=portainer-agent-stack.yml portainer
Creating network portainer_agent_network
Creating service portainer_agent
Creating service portainer_portainer
ubuntu@node1:~$
```

Figure 9.18 – Launching the Portainer Stack on our Docker Swarm cluster

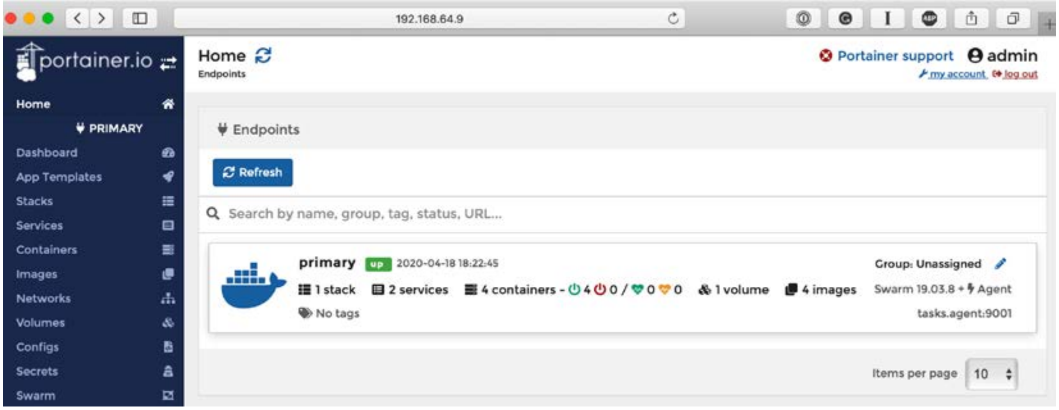
Once the stack has been created, you should be able to go to the IP address of `node1` with `:9000` at the end in your browser; for example, I opened `http://192.168.64.9:9000`.

Swarm differences

As already mentioned, there are a few changes to the Portainer interface when it is connected to a Docker Swarm cluster. In this section, we will cover them. If a part of the interface is not mentioned, then there is no difference between running Portainer in single-host mode or Docker Swarm mode. The first change we are going to look at is what changes when you first log in to the newly launched Portainer.

Endpoints

The first thing you will have to do when you log in is to select an endpoint. As you can see from the following screen, there is a single one called **primary**:



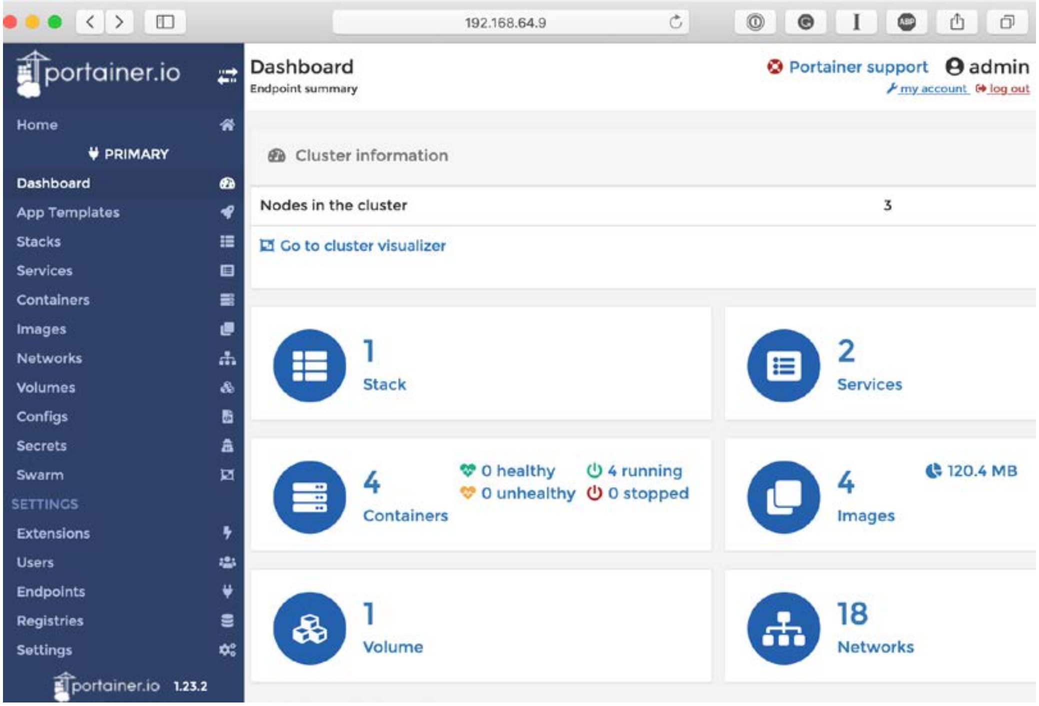
primary up 2020-04-18 18:22:45
1 stack 2 services 4 containers - 4 0 / 0 0 1 volume 4 images
No tags
Group: Unassigned
Swarm 19.03.8 + Agent
tasks.agent:9001

Figure 9.19 – Viewing the endpoint

Clicking on the endpoint will take you to the dashboard. We will look at endpoints again at the end of the section.

The dashboard and Swarm

One of the first changes you will notice is that the dashboard now displays some information on the Swarm cluster. As you can see in the following screen, there is a **Cluster information** section at the top:



Cluster information

Nodes in the cluster 3

Go to cluster visualizer

1 Stack

2 Services

4 Containers — 0 healthy, 0 unhealthy, 4 running, 0 stopped

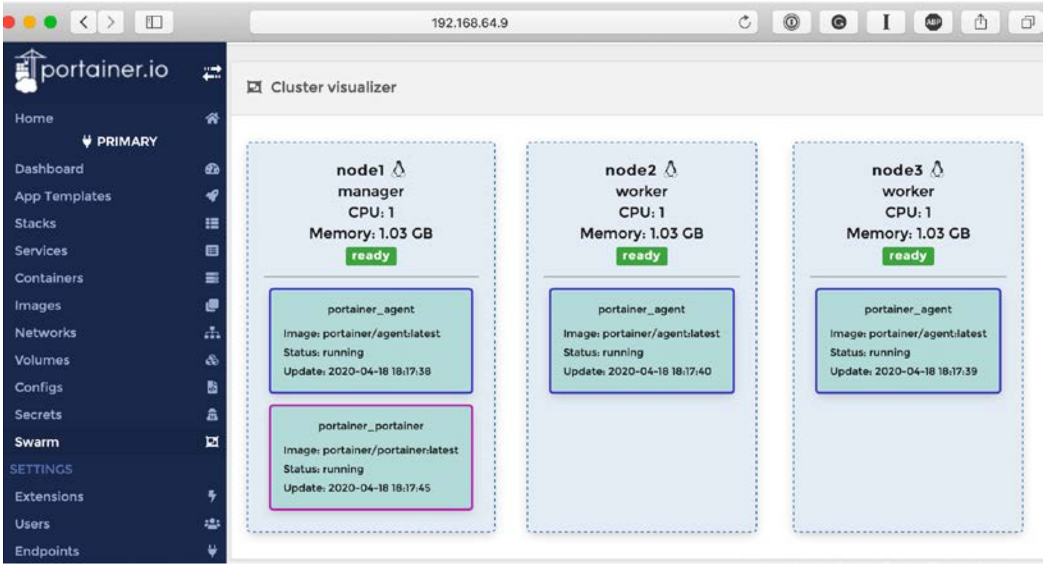
4 Images — 120.4 MB

1 Volume

18 Networks

Figure 9.20 – Getting a cluster overview

Clicking on **Go to cluster vizualizer** will take you to the Swarm page. This gives you a visual overview of the cluster, where the only running containers are currently the ones needed to provide and support the Portainer service:



node1
manager
CPU: 1
Memory: 1.03 GB
ready

portainer_agent
Image: portainer/agent:latest
Status: running
Update: 2020-04-18 18:17:38

portainer_portainer
Image: portainer/portainer:latest
Status: running
Update: 2020-04-18 18:17:45

node2
worker
CPU: 1
Memory: 1.03 GB
ready

portainer_agent
Image: portainer/agent:latest
Status: running
Update: 2020-04-18 18:17:40

node3
worker
CPU: 1
Memory: 1.03 GB
ready

portainer_agent
Image: portainer/agent:latest
Status: running
Update: 2020-04-18 18:17:39

Figure 9.21 – Visualizing the cluster

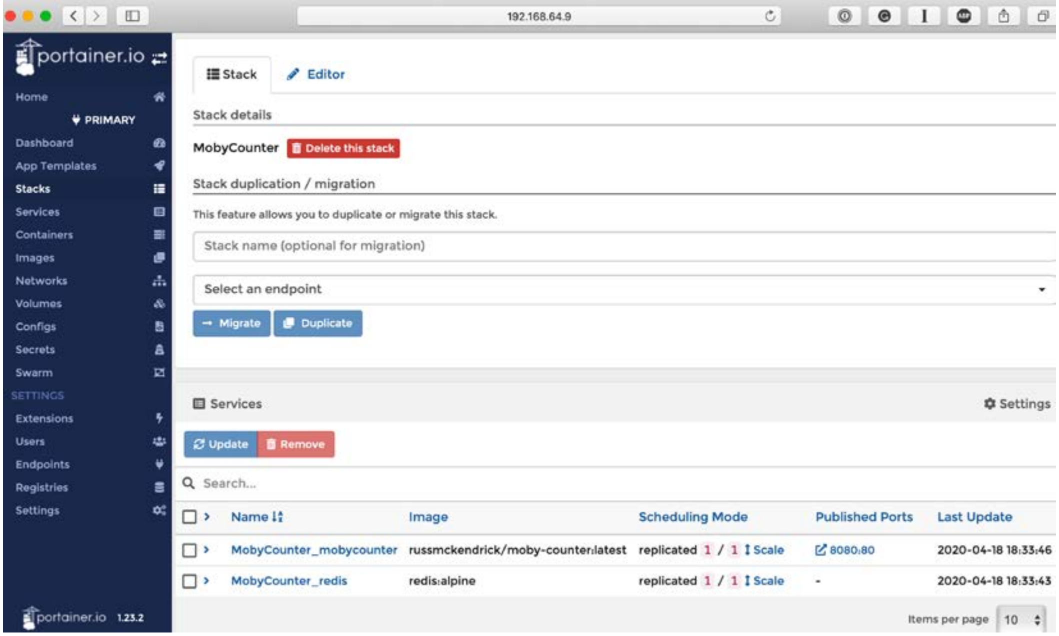
Stacks

The one item we haven't covered in the left-hand menu is **Stacks**. From here, you can launch stacks as we did when we looked at Docker Swarm. In fact, let's take the Docker Compose file we used, which looks like the following:

```
version: '3'

services:
  redis:
    image: redis:alpine
    volumes:
      - redis_data:/data
    restart: always
  mobycounter:
    depends_on:
      - redis
    image: russmckendrick/moby-counter
    ports:
      - '8080:80'
    restart: always

volumes:
  redis_data:
```

Click on the **+ Add stack** button and then paste the preceding contents into the web editor. Enter the name `MobyCounter`. Do not add any spaces or special characters to the name as this is used by Docker for. Then click on **Deploy the stack**.

Once deployed, you will be able to click on **MobyCounter** and manage the stack:



Stack details

MobyCounter

Stack duplication / migration

This feature allows you to duplicate or migrate this stack.

Stack name (optional for migration)

Select an endpoint

Services

Name	Image	Scheduling Mode	Published Ports	Last Update
MobyCounter_mobycounter	russmckendrick/moby-counter:latest	replicated 1 / 1 Scale	8080:80	2020-04-18 18:33:46
MobyCounter_redis	redis:alpine	replicated 1 / 1 Scale	-	2020-04-18 18:33:43

Items per page 10

Figure 9.22 – Launching a stack

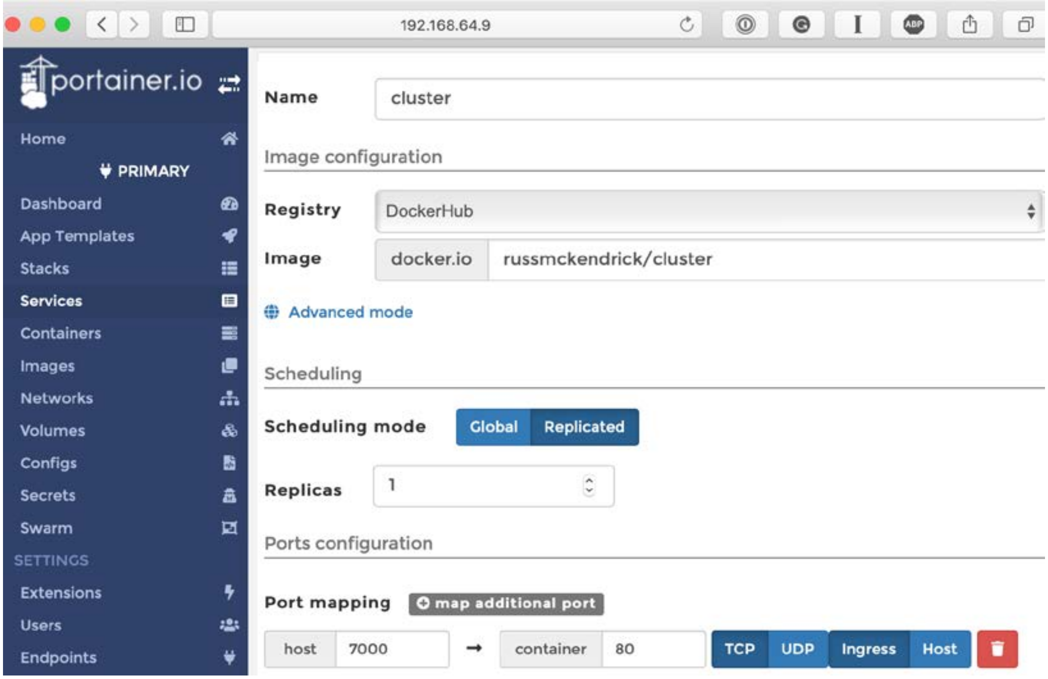
Stacks are a collection of services. Let's take a look at them next.

Services

This page is where you can create and manage services; it should already be showing several services, including Portainer. So that we don't cause any problems with the running Portainer container, we are going to create a new service. To do this, click on the **+ Add Service** button. On the page that loads, enter the following:

- **Name**: `cluster`
- **Image**: `russmckendrick/cluster`
- **Scheduling mode**: `Replicated`
- **Replicas**: `1`

This time, we need to add a port mapping for port `7000` on the host to map to port `80` to the container, this is because some of the common ports we have been using previously are already taken on the hosts due to the services and stacks we have already launched:



Name: cluster

Image configuration

Registry: DockerHub

Image: docker.io russmckendrick/cluster

Scheduling

Scheduling mode: Global [✓ Replicated]

Replicas: 1

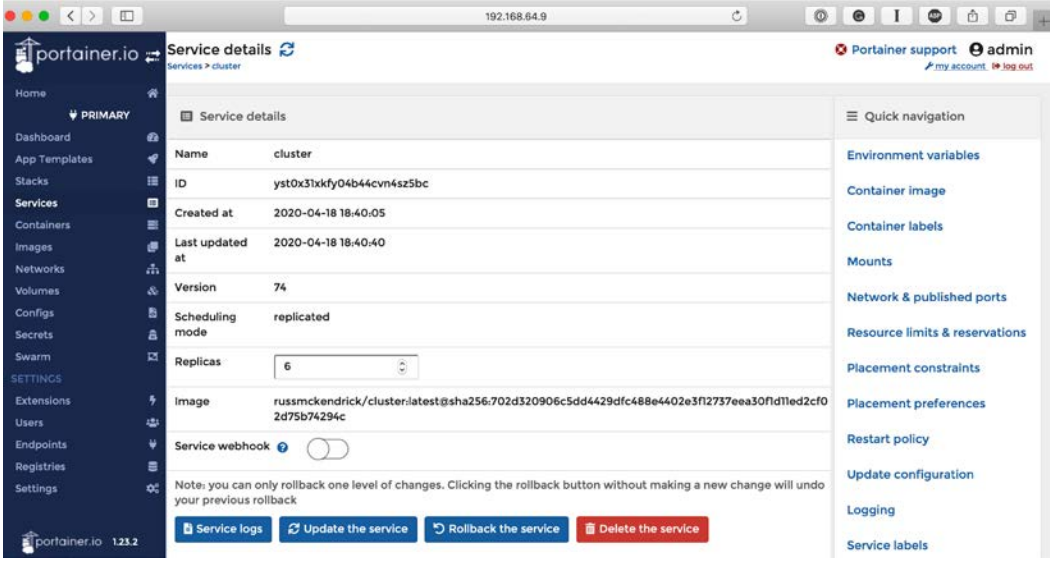
Ports configuration

Port mapping: host 7000 → container 80 [✓ TCP] UDP [✓ Ingress] Host

Figure 9.23 – Launching a service

Once you have entered the information, click on the **Create the service** button. You will be taken back to the list of services, which should now contain the cluster service we just added. You may have noticed that in the **Scheduling mode** section, there is an option to scale. Click on it and increase the number of replicas to **6** for our **cluster** service.

Clicking on **cluster** in the **Name** section takes us to an overview of the service. As you can see, there is a lot of information on the service:



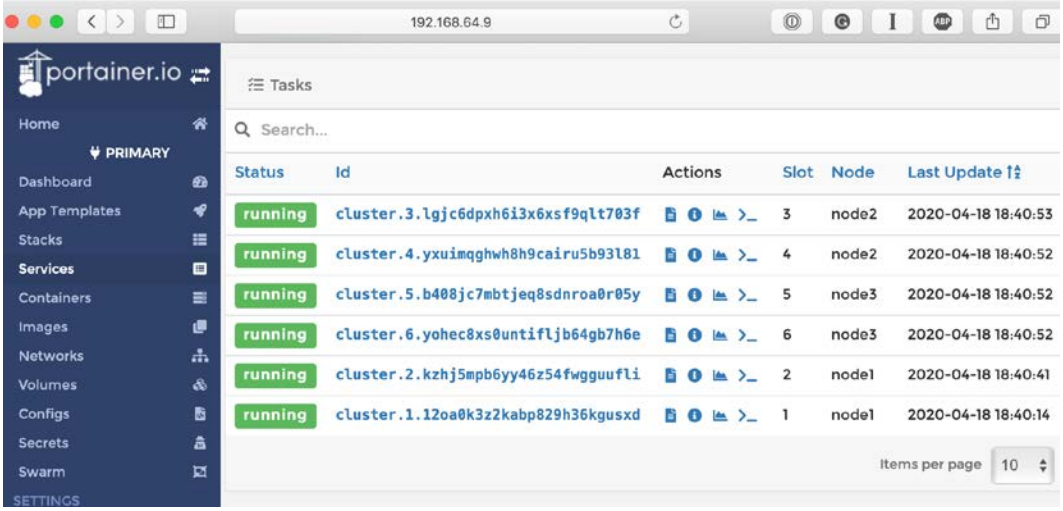
Service details

Name	cluster
ID	yst0x31xkfy04b44cvn4sz5bc
Created at	2020-04-18 18:40:05
Last updated at	2020-04-18 18:40:40
Version	74
Scheduling mode	replicated
Replicas	6
Image	russmckendrick/cluster:latest@sha256:702d320906c5dd4429dfc488e4402e3f12737eea30f1d11ed2cf02d75b74294c
Service webhook	○

Note: you can only rollback one level of changes. Clicking the rollback button without making a new change will undo your previous rollback

Figure 9.24 – Viewing details on the service

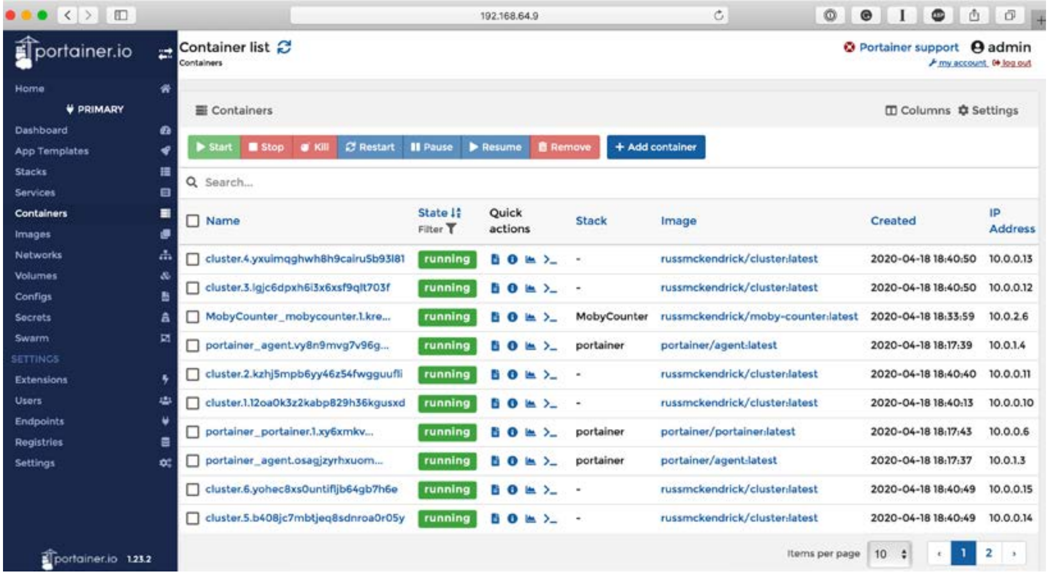
You can make a lot of changes to the service on the fly, including placement constraints, the restart policy, adding service labels, and more. Right at the bottom of the page is a list of the tasks associated with the service:



Status	Id	Actions	Slot	Node	Last Update
running	cluster.3.lgjc6dpxh6i3x6xsf9qlt703f	>_	3	node2	2020-04-18 18:40:53
running	cluster.4.yxuimqghwh8h9cairu5b93l81	>_	4	node2	2020-04-18 18:40:52
running	cluster.5.b408jc7mbtjeq8sdnroa0r05y	>_	5	node3	2020-04-18 18:40:52
running	cluster.6.yohec8xs0untifljb64gb7h6e	>_	6	node3	2020-04-18 18:40:52
running	cluster.2.kzhj5mpb6yy46z54fwgguufli	>_	2	node1	2020-04-18 18:40:41
running	cluster.1.12oa0k3z2kabp829h36kgusxd	>_	1	node1	2020-04-18 18:40:14

Figure 9.25 – Viewing the tasks

As you can see, we have six running tasks, two on each of our three nodes. Clicking on **Containers** in the left-hand menu may show something different than you expect:

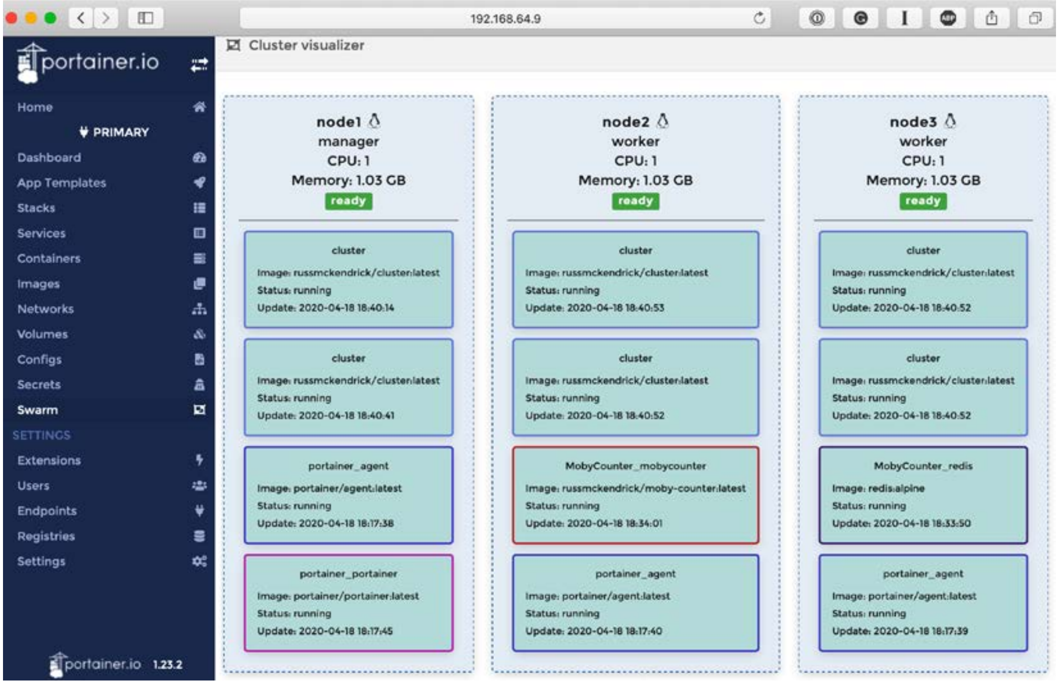


Name	State Filter	Quick actions	Stack	Image	Created	IP Address
cluster.4.yxuimqghwh8h9cairu5b93l81	running	>_	-	russmckendrick/cluster:latest	2020-04-18 18:40:50	10.0.0.13
cluster.3.lgjc6dpxh6i3x6xsf9qlt703f	running	>_	-	russmckendrick/cluster:latest	2020-04-18 18:40:50	10.0.0.12
MobyCounter_mobycounter.1.kre...	running	>_	MobyCounter	russmckendrick/moby-counter:latest	2020-04-18 18:33:59	10.0.2.6
portainer_agent.vy8n9mvg7v96g...	running	>_	portainer	portainer/agent:latest	2020-04-18 18:17:39	10.0.1.4
cluster.2.kzhj5mpb6yy46z54fwgguufli	running	>_	-	russmckendrick/cluster:latest	2020-04-18 18:40:40	10.0.0.11
cluster.1.12oa0k3z2kabp829h36kgusxd	running	>_	-	russmckendrick/cluster:latest	2020-04-18 18:40:13	10.0.0.10
portainer_portainer.1.xy6xmkv...	running	>_	portainer	portainer/portainer:latest	2020-04-18 18:17:43	10.0.0.6
portainer_agent.osagjzyrhxuom...	running	>_	portainer	portainer/agent:latest	2020-04-18 18:17:37	10.0.1.3
cluster.6.yohec8xs0untifljb64gb7h6e	running	>_	-	russmckendrick/cluster:latest	2020-04-18 18:40:49	10.0.0.15
cluster.5.b408jc7mbtjeq8sdnroa0r05y	running	>_	-	russmckendrick/cluster:latest	2020-04-18 18:40:49	10.0.0.14

Figure 9.26 – Listing all of the containers

You can see all of the containers that are running within the cluster, rather than just the ones on the node where we deployed Portainer. You may recall that when we looked at the cluster visualizer, there were Portainer Agent containers running on each of the nodes within the cluster that were launched as part of the stack. These are feeding information back, giving us an overall view of our cluster.

Going back to the cluster visualizer now shows us that there are a lot more containers running:



node1
manager
CPU: 1
Memory: 1.03 GB

cluster
Image: russmckendrick/cluster:latest
Status: running
Update: 2020-04-18 18:40:14

cluster
Image: russmckendrick/cluster:latest
Status: running
Update: 2020-04-18 18:40:41

portainer_agent
Image: portainer/agent:latest
Status: running
Update: 2020-04-18 18:17:38

portainer_portainer
Image: portainer/portainer:latest
Status: running
Update: 2020-04-18 18:17:45

node2
worker
CPU: 1
Memory: 1.03 GB

cluster
Image: russmckendrick/cluster:latest
Status: running
Update: 2020-04-18 18:40:53

cluster
Image: russmckendrick/cluster:latest
Status: running
Update: 2020-04-18 18:40:52

MobyCounter_mobycounter
Image: russmckendrick/moby-counter:latest
Status: running
Update: 2020-04-18 18:34:01

portainer_agent
Image: portainer/agent:latest
Status: running
Update: 2020-04-18 18:17:40

node3
worker
CPU: 1
Memory: 1.03 GB

cluster
Image: russmckendrick/cluster:latest
Status: running
Update: 2020-04-18 18:40:52

cluster
Image: russmckendrick/cluster:latest
Status: running
Update: 2020-04-18 18:40:52

MobyCounter_redis
Image: redis:alpine
Status: running
Update: 2020-04-18 18:33:50

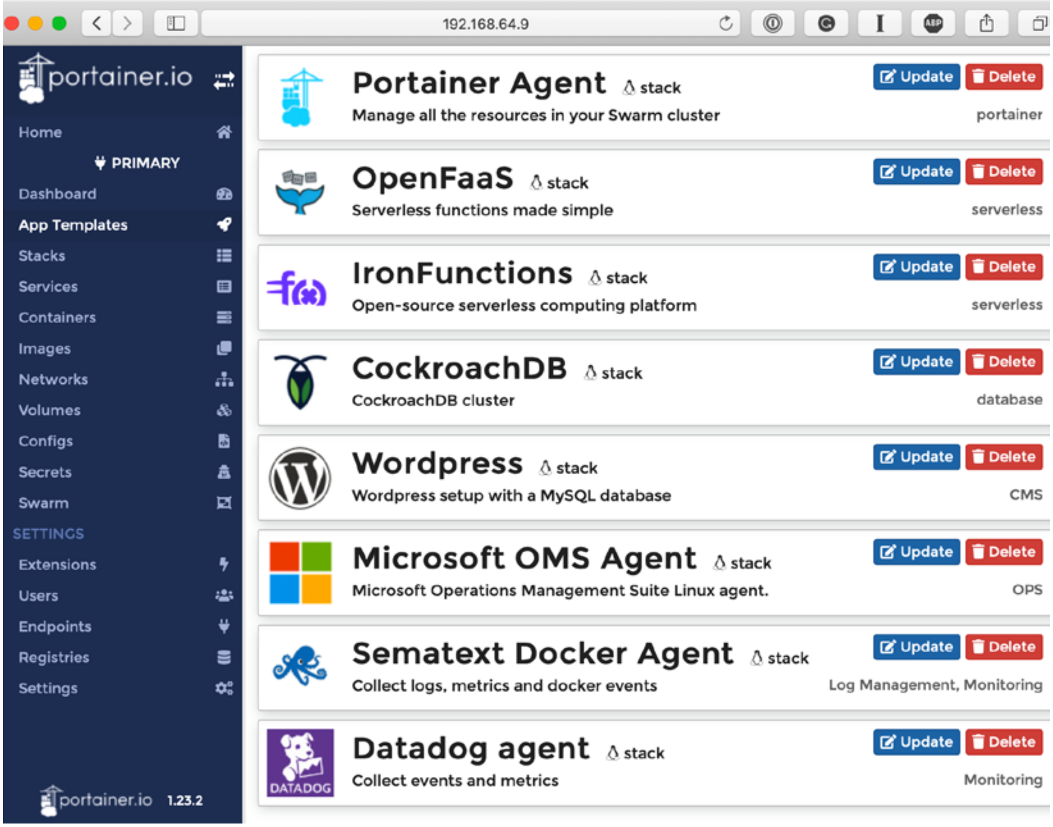
portainer_agent
Image: portainer/agent:latest
Status: running
Update: 2020-04-18 18:17:39

Figure 9.27– Viewing the visualizer

Let's take a look at what else has changed now we have moved to running Docker Swarm.

App Templates

Going to the **App Templates** page now shows stacks instead of containers:



Portainer Agent stack
Manage all the resources in your Swarm cluster
portainer

OpenFaaS stack
Serverless functions made simple
serverless

IronFunctions stack
Open-source serverless computing platform
serverless

CockroachDB stack
CockroachDB cluster
database

Wordpress stack
Wordpress setup with a MySQL database
CMS

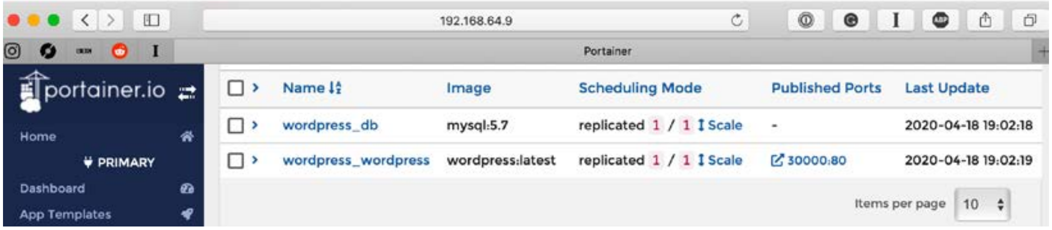
Microsoft OMS Agent stack
Microsoft Operations Management Suite Linux agent.
OPS

Sematext Docker Agent stack
Collect logs, metrics and docker events
Log Management, Monitoring

Datadog agent stack
Collect events and metrics
Monitoring

Figure 9.28 – Viewing the stack templates

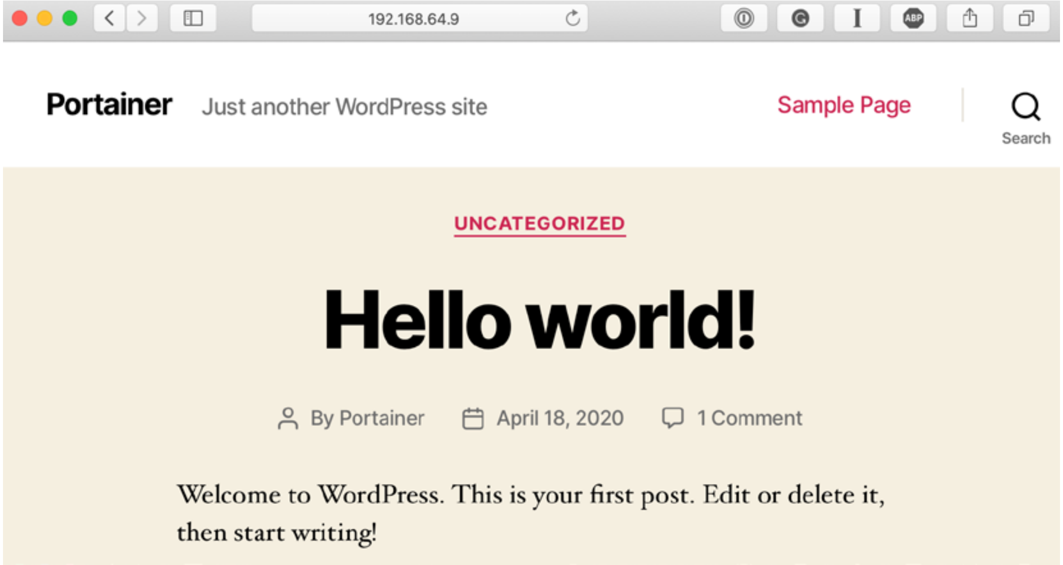
As you can see, there are quite a few defaults listed, clicking on one, such as **Wordpress** will take you to a page where you simply have to enter a few details and then click on the **Deploy the stack** button. Once deployed, you should be able to then go to the **Stacks** page and view the port that has been assigned to the service:



	Name	Image	Scheduling Mode	Published Ports	Last Update
□	wordpress_db	mysql:5.7	replicated 1 / 1 Scale	-	2020-04-18 19:02:18
□	wordpress_wordpress	wordpress:latest	replicated 1 / 1 Scale	30000:80	2020-04-18 19:02:19

Figure 9.29 – Launching Wordpress

Once you know the port, entering the IP address of any of your nodes and the port will take you to the application, which, after following the installation instructions, looks like the following:



Hello world!

By Portainer April 18, 2020 1 Comment

Welcome to WordPress. This is your first post. Edit or delete it, then start writing!

Figure 9.30 – WordPress up and running

These templates are hosted on GitHub and you can find a link in the *Further reading* section.

Removing the cluster

Once you have finished exploring Portainer on Docker Swarm, you can remove the cluster by running the following commands on your local machine:

```
$ multipass delete --purge node1
$ multipass delete --purge node2
$ multipass delete --purge node3
```

It is important to remove your running nodes on your local machine, because if you don't, they will continue to run and consume resources.

Summary

That concludes our deep dive with Portainer. As you can see, Portainer is very powerful, yet simple to use, and will only continue to grow and integrate more of the Docker ecosystem as features are released. With Portainer, you can do a lot of manipulation with not only your hosts but also the containers and services running on single or cluster hosts.

In the next chapter, we are going to look at another container clustering solution supported by Docker called Kubernetes.

Questions

1. On a macOS or Linux machine, what is the path to mount the Docker socket file?
2. What is the default port Portainer runs on?
3. True or false: You can use Docker Compose files as application templates.
4. True or false: The stats shown in Portainer are only real time, you can't view historical data.

Further reading

You can find more information on Portainer here:

1. Main website: `https://www.portainer.io`
2. Portainer on GitHub: `https://github.com/portainer/`
3. Latest documentation: `https://portainer.readthedocs.io/en/latest/index.html`
4. Template documentation: `https://portainer.readthedocs.io/en/latest/templates.html`
5. Templates: `https://github.com/portainer/templates`

10
Running Docker in Public Clouds

So far, we have been using Digital Ocean to launch containers on a cloud-based infrastructure. In this chapter, we will look at the container solutions offered by **Amazon Web Services** (**AWS**), Microsoft Azure, and Google Cloud.

Before we talk about these container services, we will also go into a little bit of the history behind each one of the cloud providers and cover the installation of any of the command-line tools required to launch the container services.

The following topics will be covered in this chapter:

- AWS
- Microsoft Azure
- Google Cloud Run

Technical requirements

In this chapter, we will be using AWS, Microsoft Azure, and Google Cloud, so if you are following along, you will need active accounts with one or all of them.

Don't forget that if you are following along, there may be charges for the services you launch in public cloud providers – please ensure that you terminate resources when you have finished with them to avoid any unexpected costs.

Amazon Web Services

The first of the public cloud providers we are going to be looking at in this chapter is AWS. It was first launched in July 2002 as an internal service used within Amazon to provide a few disparate services to support the Amazon retail site. A year or so later, an internal presentation at Amazon laid the groundwork for what AWS was to become: a standardized and completely automated compute infrastructure to support Amazon's vision of a web-based retail platform.

At the end of the presentation, it was mentioned that Amazon could possibly sell access to some of the services AWS had to offer to help fund the infrastructure investment required to get the platform off the ground. In late 2004, the first of these public services was launched – **Amazon Simple Queue Service** (**Amazon SQS**), a distributed message queuing service. Around this time, Amazon started work on services that it could consume for the retail site and services it could sell to the public.

In 2006, AWS was relaunched, and Amazon SQS was joined by **Amazon Simple Storage Service** (**Amazon S3**) and **Amazon Elastic Compute Cloud** (**Amazon EC2**), and from there, AWS's popularity grew. At the time of writing, AWS has an annual revenue of over $25 billion and the number of services offered as part of the platform has grown from the original 3 to over 210.

These services include **Amazon Elastic Container Registry** (**Amazon ECR**), which we looked at in *Chapter 3*, *Storing and Distributing Images*, **Amazon Elastic Container Service** (**Amazon ECS**), and **AWS Fargate**. While these last two services are actually designed to work together, let's take a quick look at what options were available for using Amazon ECS before Amazon introduced AWS Fargate.

Amazon ECS – EC2-backed

Amazon ECS is described by Amazon as follows:

> *"A highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on a cluster."*

In essence, it allows you to manage the placement and availability of Docker containers on a cluster of compute resources before you launch your first ECS cluster. Let's quickly have a look at some of the terminology involved and get an idea of how it is going to hang together. When we launch our Amazon ECS-hosted application, we will need to define the following:

- Containers
- Tasks
- Services
- Clusters

The following diagram will give you an idea of how the preceding elements are going to be utilized within our cluster; you can clearly see the containers, tasks, and the service, which in this diagram, is the box that spans the three instances:

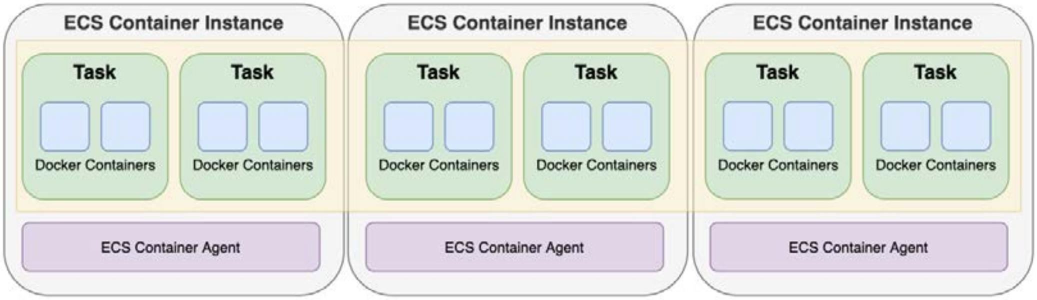
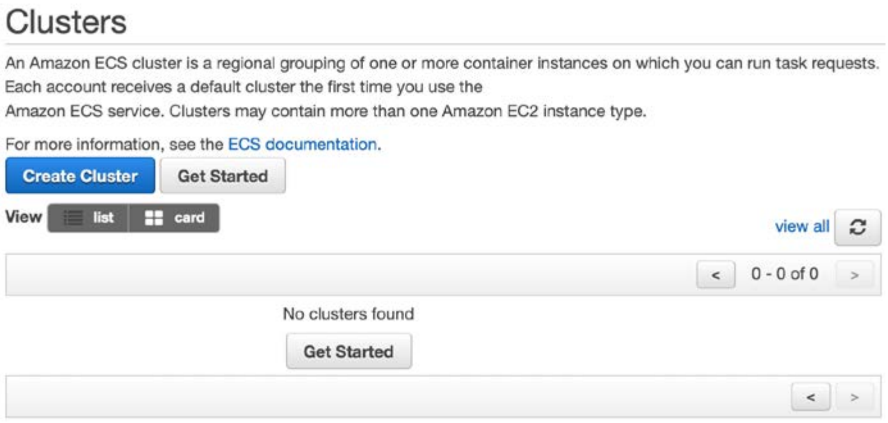


Figure 10.1 – Amazon ECS container instances

The important difference between the preceding diagram and if we were going to be using AWS Fargate is the fact that we are running container instances. To launch a cluster of EC2 instances, do the following:

1. Open up the AWS console at `https://console.aws.amazon.com/`.
2. Sign in.
3. Select **ECS** in the **Services** menu in the top-left corner of the page.
4. Once you have the Amazon ECS page open, select your preferred region in the region switcher in the top-right corner – as I am based in the UK, I have chosen **London**.

5. Click on **Clusters**, which is the first entry in the left-hand menu:



Clusters

An Amazon ECS cluster is a regional grouping of one or more container instances on which you can run task requests. Each account receives a default cluster the first time you use the Amazon ECS service. Clusters may contain more than one Amazon EC2 instance type.

For more information, see the ECS documentation.

No clusters found

Figure 10.2 – Creating Amazon ECS Clusters

6. Click the **Create Cluster** button and you will be presented with three options:

   Networking only

   EC2 Linux + Networking

   EC2 Windows + Networking

7. As we are going to be launching an AWS Fargate cluster next and our test application is Linux-based, select **EC2 Linux + Networking** and then on the **Next Step** button at the bottom of the options, and you should find yourself on the **Configure cluster** page.

8. We need to name our cluster. A cluster name is a unique name; I have called mine `RussTestECSCluster`.

9. Leave the **Create an empty cluster** option unticked. As we are wanting to launch a test cluster, we are going to tweak some of the settings to make our cluster a little more affordable, so in the instance configuration, we are going to enter the following:

   Provisioning Model: Rather than the selected by default **On-Demand Instance** option, select **Spot**, and as you can see from the descriptive text, we can get up a 90% saving on the on-demand prices here. Selecting **Spot** will also change the options listed below **Provisioning Model**.

   Spot Instance allocation strategy: Select **Lowest Price** as we don't really care about making our test cluster highly available.

   EC2 instance types: Select **m4.large** from the drop-down list.

Maximum price (per instance/hour): Here, we need to enter the maximum price we want to pay for the instance – as our cluster won't be up for long, I left this empty; however, feel free to view the **Spot prices** and **On-Demand prices** and enter your own maximum price.

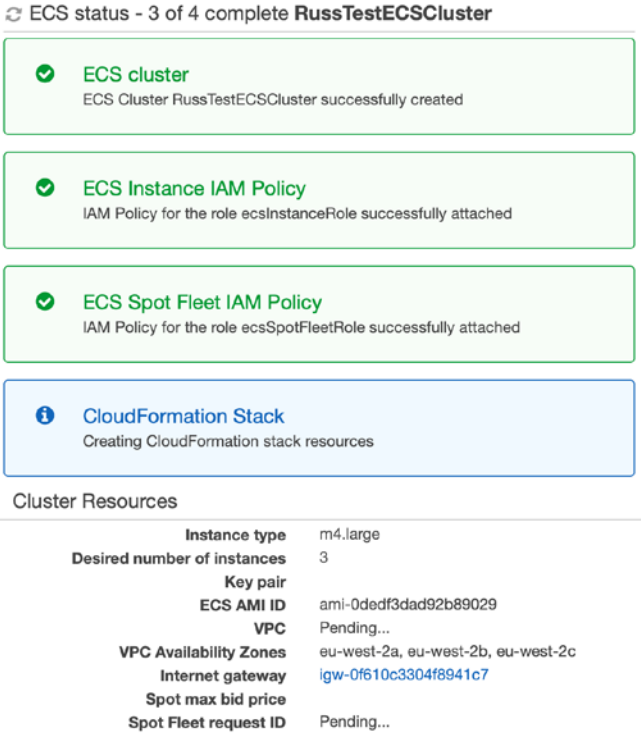
Number of instances: I changed this from `1` to `3`; when using spot instances, it is important that you have more than 1 instance to maintain a basic level of availability.

EC2 Ami Id: This option can't be changed.

EBS storage (GiB): I left this at the default, **22GB**.

Key pair: I left this as **None** - **Unable to SSH**.

10. For **Networking**, **Container instance IAM role**, **Spot Fleet IAM role**, and **Tags**, I left everything at their defaults before finally reaching the very last option, **CloudWatch Container Insights**, I ticked the box for **Enable Container Insights**. Once everything's filled in, click on the **Create** button. Once clicked, your cluster will be launched, and you can track the progress as resources are configured and launched:



ECS status - 3 of 4 complete **RussTestECSCluster**

ECS cluster
ECS Cluster RussTestECSCluster successfully created

ECS Instance IAM Policy
IAM Policy for the role ecsInstanceRole successfully attached

ECS Spot Fleet IAM Policy
IAM Policy for the role ecsSpotFleetRole successfully attached

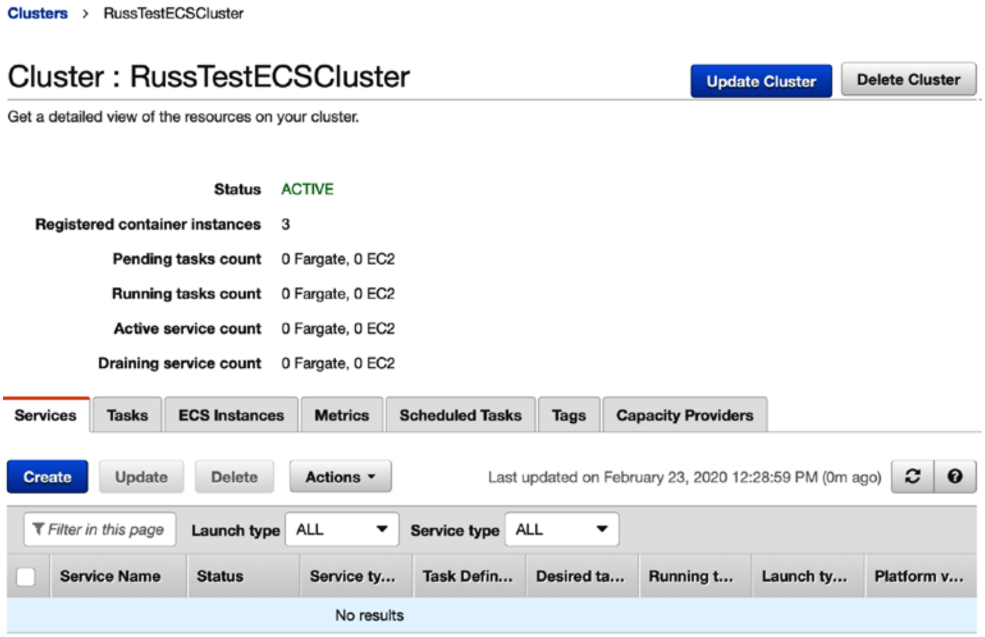
CloudFormation Stack
Creating CloudFormation stack resources

Cluster Resources

Instance type	m4.large
Desired number of instances	3
Key pair	
ECS AMI ID	ami-0dedf3dad92b89029
VPC	Pending...
VPC Availability Zones	eu-west-2a, eu-west-2b, eu-west-2c
Internet gateway	igw-0f610c3304f8941c7
Spot max bid price	
Spot Fleet request ID	Pending...

Figure 10.3 – Launching the ECS cluster

11. Once it's launched, you can click on the **View cluster** button, which will take you the following screen:



Cluster : RussTestECSCluster

Get a detailed view of the resources on your cluster.

Status	ACTIVE
Registered container instances	3
Pending tasks count	0 Fargate, 0 EC2
Running tasks count	0 Fargate, 0 EC2
Active service count	0 Fargate, 0 EC2
Draining service count	0 Fargate, 0 EC2

Last updated on February 23, 2020 12:28:59 PM (0m ago)

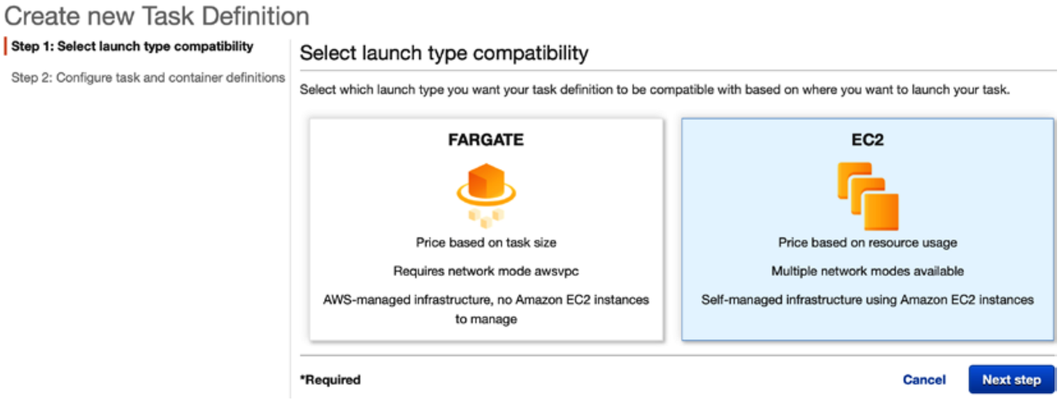
Launch type ALL

Service type ALL

□	Service Name	Status	Service ty...	Task Defin...	Desired ta...	Running t...	Launch ty...	Platform v...
			No results					

Figure 10.4 – Looking at the ECS cluster

12. Now that we have a cluster, we need to create a task definition. To do this, click on **Task Definitions** in the left-hand menu and then click on the **Create new Task Definition** button. The first option we are presented with asks which launch type we want to use. We have just created an EC2-based cluster, so select **EC2**:


Step 1: Select launch type compatibility
Step 2: Configure task and container definitions
Cancel
Next step

Create new Task Definition

Select launch type compatibility

Select which launch type you want your task definition to be compatible with based on where you want to launch your task.

FARGATE	EC2
Price based on task size	Price based on resource usage
Requires network mode awsvpc	Multiple network modes available
AWS-managed infrastructure, no Amazon EC2 instances to manage	Self-managed infrastructure using Amazon EC2 instances

*Required

Figure 10.5 – Select which type of task we need

13. Once selected, click on the **Next step** button and you will be taken to the **Configure task and container definitions** page. This is another form – here, enter the following:

    Task Definition Name: Enter `cluster-task`.

    Requires Compatibilities: This is set at **EC2** and can not be changed.

    Task Role: Leave this as **None**.

    Network Mode: Leave this at **<default>**.

    Task execution IAM role: If you don't see **ecsTaskExecutionRole** already listed, select **Create new role**.

    Task memory (MiB): Enter `1gb`.

    Task CPU (unit): Enter `1 vcpu`.

14. This brings us down to the container definitions. As a task definition is a collection of containers, you can enter multiple containers here. In this example, we are going to add a single container. To start, click on **Add container** – this will slide in, yes you have a guessed it, another form from the right-hand side of the screen. Here, enter the following:

    Container name: Enter `cluster-container`.

    Image: Enter `russmckendrick/cluster:latest`.

    Private repository authentication: Leave unticked.

    Memory Limits (MiB): Add a hard limit of `128`.

    Port mappings: Enter `80` for the **Host** and **Container** ports and leave **Protocol** as `tcp`.

15. Leave the remainder of the form at the default settings and then click on **Add**.

16. The final part of the task definition we need to define is **Volumes**. Scroll down to the bottom and click on the **Add volume** button and enter just the name, `cluster-task`, leave **Volume type** as **Bind month** and **Source path** empty, and then click on the **Add** button. Now, click on **Create** and then return to the cluster via the left-hand **Clusters** menu.

17. Now that we have our cluster, and our task definition, we can create a service. To do this within the **Services** tab of the cluster overview, just click on the **Add** button in the:

    Launch type: Select **EC2**.

    Task Definition: Make sure that **cluster-task** is selected.

    Cluster: This should default to your cluster.

    Service name: Enter `cluster-service`.

    Service Type: Leave this as **REPLICA**.

    Number of tasks: Enter 2.

    Minimum healthy percent: Leave at **100**.

    Maximum percent: Leave at **200**.

18. Leave the **Deployments** and **Task Placement** options at the defaults and click on the **Next step** button at the bottom of the page, which will take you to the **Configure network** page:

    Load balancer type: Select **None**.

    Enable service discovery integration: Untick this option.

19. Click on **Next step** and leave **Service Auto Scaling** on the default **Do not adjust the service's desired count** option, and then click on **Next Step**. After reviewing the options, click on the **Create Service** button.

20. Before we open our running container, we need to open the firewall rules. To do this, select **VPC** from the **Services** menu, and then from the left-hand menu, select **Security Groups**. Once on the security group page, highlight the **EC2ContainerService** entry, and then click on the **Inbound Rules** tab. Once there, click the **Edit Rules** button and enter the following:

    Type: **Custom TCP Rule**.

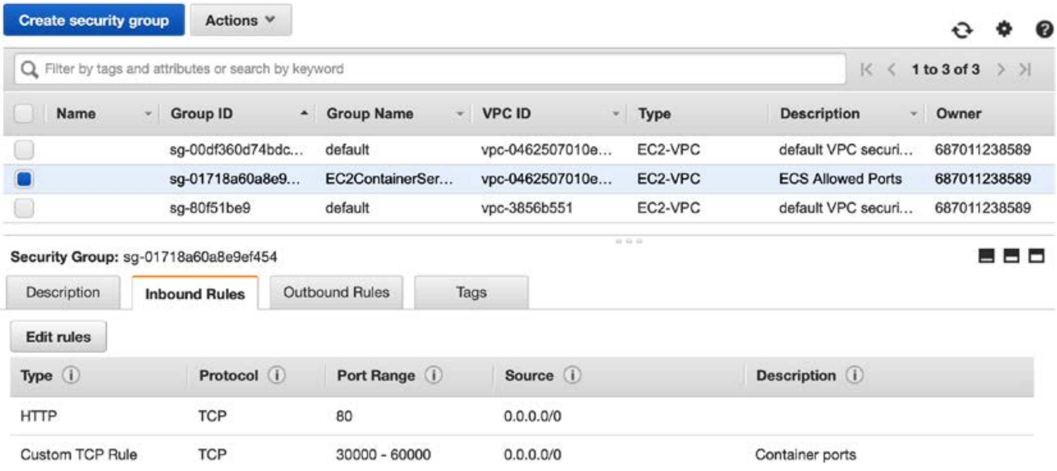
    Protocol: **TCP**.

    Port Range: Enter `30000-60000`.

    Source: Enter `0.0.0.0/0`.

    Description: Enter `Container ports`.

21. Then, click on the **Save Rules** button. This should leave you with something like the following:



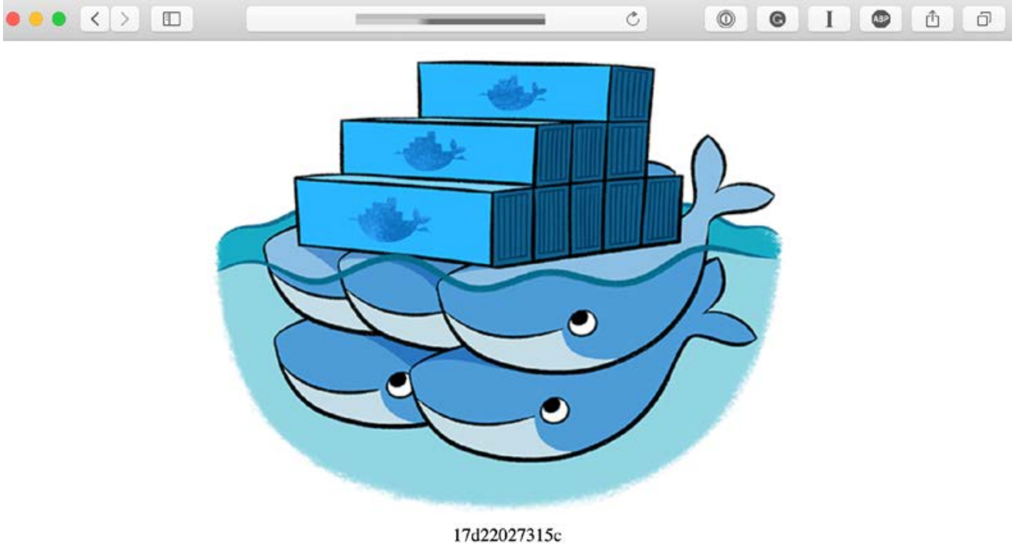
	Name	Group ID	Group Name	VPC ID	Type	Description	Owner
□		sg-00df360d74bdc...	default	vpc-0462507010e...	EC2-VPC	default VPC securi...	687011238589
☑		sg-01718a60a8e9...	EC2ContainerSer...	vpc-0462507010e...	EC2-VPC	ECS Allowed Ports	687011238589
□		sg-80f51be9	default	vpc-3856b551	EC2-VPC	default VPC securi...	687011238589

Security Group: sg-01718a60a8e9ef454

Type	Protocol	Port Range	Source	Description
HTTP	TCP	80	0.0.0.0/0	
Custom TCP Rule	TCP	30000 - 60000	0.0.0.0/0	Container ports

Figure 10.6 – Updating the security group rules

22. Once you have set the rule, return to your ECS cluster, click on the **Tasks** tab, then select one of the two running tasks. Once the task overview place loads, scroll down to the container and extend the container listed in the table, and then click on the external link. This will take you straight to the running container:



17d22027315c

Figure 10.7 – Our working application

Now that you have seen the running container, let's remove the cluster. To do this, go to the cluster overview page and click on the **Delete Cluster** button in the top-right corner – from there, follow the on-screen instructions.

Now that was the old way of launching containers sort-of nativity in AWS; I say sort of because we are still starting EC2 instances to power our cluster, and even though we used spot instances, we had a lot of unused resources that we would have been paying for – what if we could just launch containers and not have worry about managing a bunch of EC2 resources?

Amazon ECS – AWS Fargate backed

In November 2017, Amazon announced that they had been working on AWS Fargate – this is a service that is Amazon ECS-compatible and removes the need to launch and manage EC2 instances. Instead, your containers are launched on Amazon's backplane, and you get per-second billing, meaning you only pay for the vCPU and memory that your containerised application requests during its life cycle.

We are going to cheat slightly and work through the Amazon ECS first-run process. You can access this by going to the following URL: `https://console.aws.amazon.com/ecs/home?#/firstRun`.

This will take us through the four steps we need to take to launch a container within a Fargate cluster. The first step in launching our AWS Fargate-hosted container is to configure the container and task definitions:

1. For our example, there are three predefined options and a custom option. Click on the **Configure** button in the custom options and enter the following information:

   Container name: `cluster-container`.

   Image: `russmckendrick/cluster:latest`.

   Memory Limits (MiB): Leave at the default.

   Port mappings: Enter `80` and leave `tcp` selected.

2. Then, click on the **Update** button. For the task definition, click on the **Edit** button and enter the following:

   Task definition name: `cluster-task`.

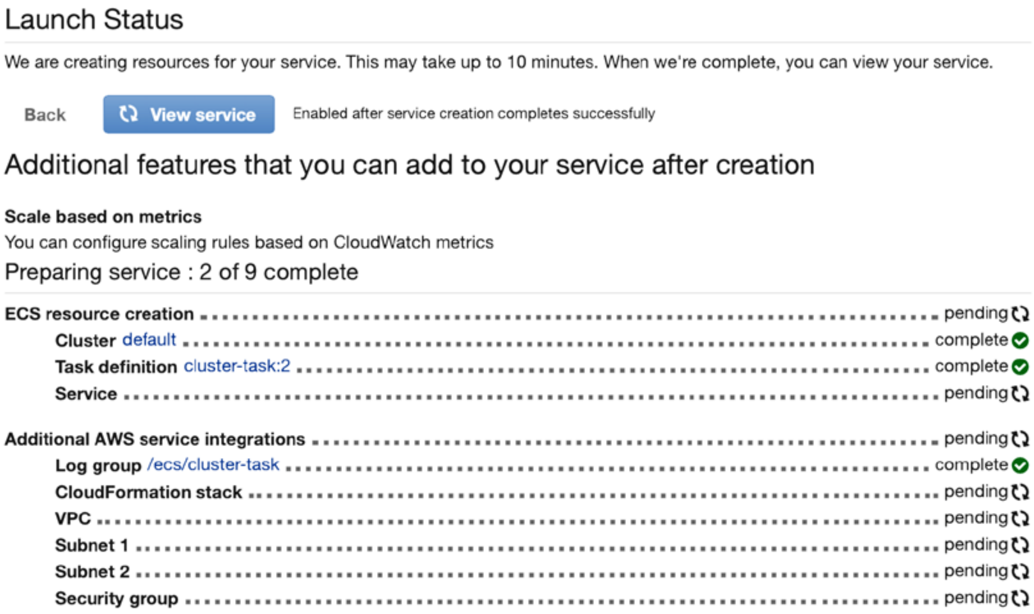
   Network mode: Should be `awsvpc`; you can't change this option.

   Task execution role: Leave as `ecsTaskExecutionRole`.

Compatibilities: This should default to `FARGATE` and you should not be able to edit it.

Task memory and Task CPU: Leave both at their default options.

3. Once everything's updated, click on the **Save** button. Now, you can click on the **Next** button at the bottom of the page. This will take us to the second step, which is where the service is defined.
4. As we discussed in the previous section, the service runs tasks, which in turn have a container associated with them. The default services are fine, so click on the **Next** button to proceed to the third step of the launch process. The first step is where the cluster is created. Again, the default values are fine, so click on the **Next** button to be taken to the review page.
5. This is your last chance to double-check the task, service, and cluster definitions before any services are launched. If you are happy with everything, click on the **Create** button. From here, you will be taken to a page where you can view the status of the various AWS services that make our AWS Fargate cluster:



Launch Status

We are creating resources for your service. This may take up to 10 minutes. When we're complete, you can view your service.

Enabled after service creation completes successfully

Additional features that you can add to your service after creation

Scale based on metrics

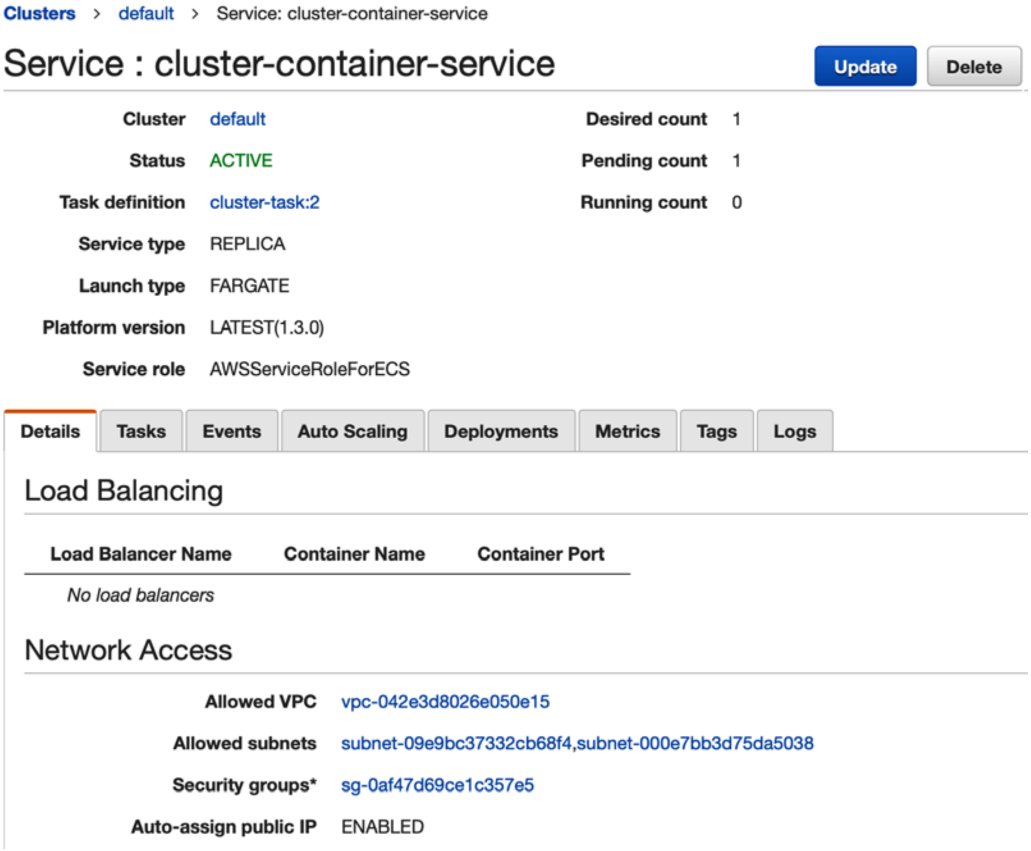
You can configure scaling rules based on CloudWatch metrics

Preparing service : 2 of 9 complete

```
ECS resource creation ........................................ pending ↻
    Cluster default .......................................... complete ✓
    Task definition cluster-task:2 ........................... complete ✓
    Service .................................................. pending ↻

Additional AWS service integrations .......................... pending ↻
    Log group /ecs/cluster-task .............................. complete ✓
    CloudFormation stack ..................................... pending ↻
    VPC ...................................................... pending ↻
    Subnet 1 ................................................. pending ↻
    Subnet 2 ................................................. pending ↻
    Security group ........................................... pending ↻
```

Figure 10.8 – Launching our Fargate cluster

6. Once everything has changed from **pending** to **complete**, you will be able to click on the **View service** button to be taken to the **Service** overview page:



Service : cluster-container-service

Cluster	default	Desired count	1
Status	ACTIVE	Pending count	1
Task definition	cluster-task:2	Running count	0
Service type	REPLICA		
Launch type	FARGATE		
Platform version	LATEST(1.3.0)		
Service role	AWSServiceRoleForECS		

Load Balancing

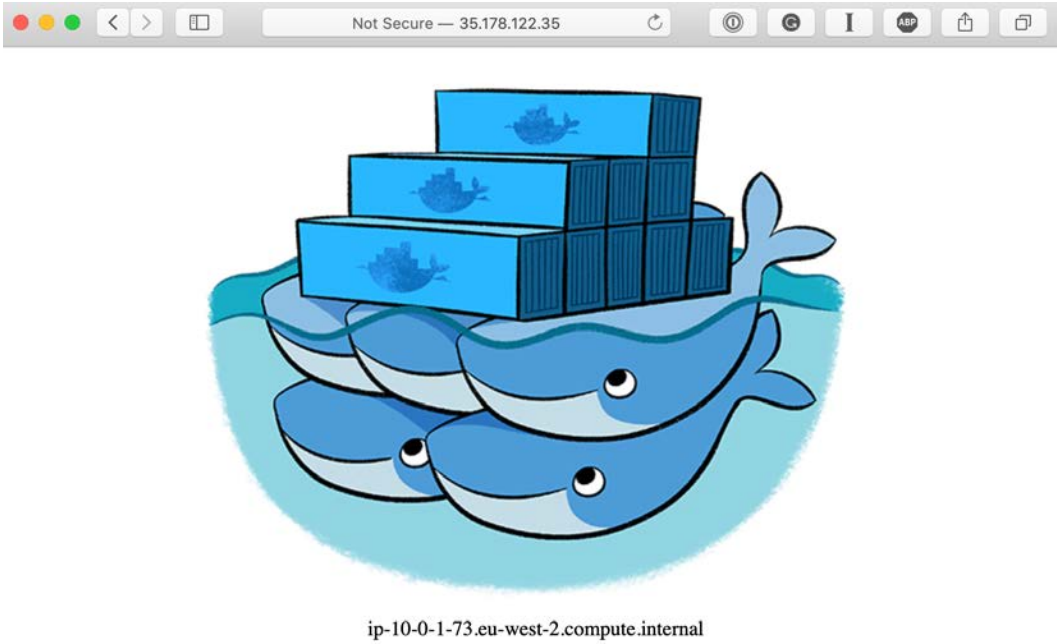
Load Balancer Name	Container Name	Container Port
No load balancers		

Network Access

Allowed VPC	vpc-042e3d8026e050e15
Allowed subnets	subnet-09e9bc37332cb68f4,subnet-000e7bb3d75da5038
Security groups*	sg-0af47d69ce1c357e5
Auto-assign public IP	ENABLED

Figure 10.9 – Looking at our Fargate cluster

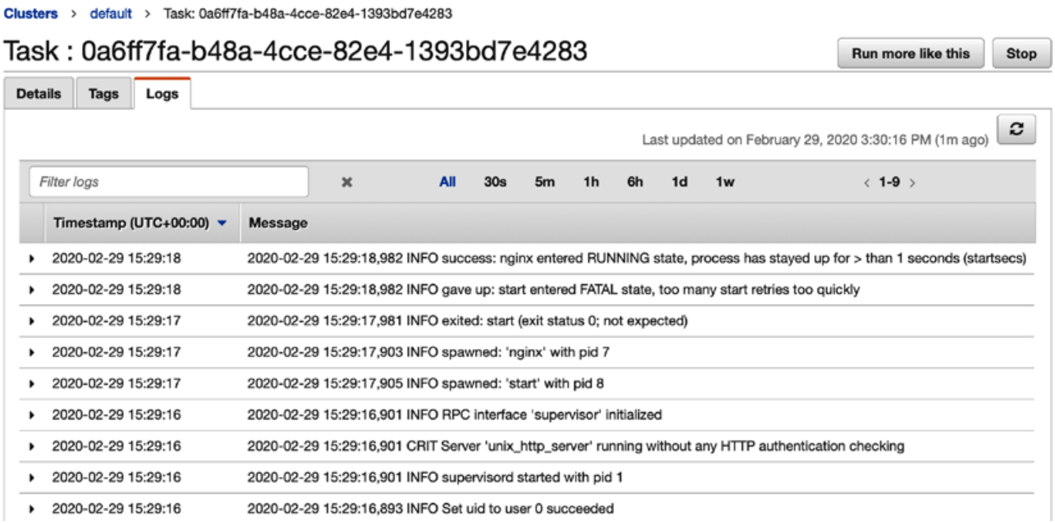
7. Now, we need to know the public IP address of our container. To find this, click on the **Task** tab, and then select the unique ID of the running task. In the **Network** section of the page, you should be able to find both the private and public IP addresses of the tasks. Entering the public IP in your browser should bring up the now-familiar cluster application:



ip-10-0-1-73.eu-west-2.compute.internal

Figure 10.10 – Our working application

You will notice that the container name that's displayed is the hostname of the container, and includes the internal IP address. You can also view the logs from the container by click on the **Logs** tab:



Task : 0a6ff7fa-b48a-4cce-82e4-1393bd7e4283

Last updated on February 29, 2020 3:30:16 PM (1m ago)

```
Timestamp (UTC+00:00)  Message
2020-02-29 15:29:18    2020-02-29 15:29:18,982 INFO success: nginx entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
2020-02-29 15:29:18    2020-02-29 15:29:18,982 INFO gave up: start entered FATAL state, too many start retries too quickly
2020-02-29 15:29:17    2020-02-29 15:29:17,981 INFO exited: start (exit status 0; not expected)
2020-02-29 15:29:17    2020-02-29 15:29:17,903 INFO spawned: 'nginx' with pid 7
2020-02-29 15:29:17    2020-02-29 15:29:17,905 INFO spawned: 'start' with pid 8
2020-02-29 15:29:16    2020-02-29 15:29:16,901 INFO RPC interface 'supervisor' initialized
2020-02-29 15:29:16    2020-02-29 15:29:16,901 CRIT Server 'unix_http_server' running without any HTTP authentication checking
2020-02-29 15:29:16    2020-02-29 15:29:16,901 INFO supervisord started with pid 1
2020-02-29 15:29:16    2020-02-29 15:29:16,893 INFO Set uid to user 0 succeeded
```

Figure 10.11 – Viewing the container logs

So, how much is this costing? To be able to run the container for an entire month would cost around $14, which works out at about $0.019 per hour.

This costing means that if you are going to be running a number of tasks 24/7, then Fargate may not be the most cost-effective way of running your containers. Instead, you may want to take the Amazon ECS EC2 option, where you can pack more containers onto your resource, or the Amazon EKS service, which we will look at later in this chapter. However, for quickly bringing up a container and then terminating it, Fargate is excellent—there is a low barrier to launching the containers and the number of supporting resources is small.

Once you have finished with your Fargate container, you should delete the cluster. This will remove all of the services associated with the cluster. Once the cluster has been removed, go into the **Task Definitions** page and deregister them if needed.

Summing up AWS

In this section of the chapter, we have only touched on how Amazon ECS works; what we haven't covered is some of the close integration that an Amazon ECS-managed container has with other AWS services, such as **Elastic Load Balancing**, **Amazon Cloud Map**, and **AWS App Mesh**. Also, using the Amazon ECS command-line tool, you can launch your Docker Compose files into an Amazon ECS-managed cluster.

Now that we have our AWS basics in place, let's move on to one of its major competitors, Microsoft Azure.

Microsoft Azure

Microsoft Azure, or Windows Azure as it started life as, is Microsoft's entry into the public cloud. It offers a mixture of the **software as a service** (**SaaS**), **platform as a service** (**PaaS**), and **infrastructure as a service** (**IaaS**) services. It began life as an internal Microsoft project with the codename of *Project Red Dog* around 2005. Project Red Dog was a continuation of the *Red Dog OS*, which was a fork of the Windows operating system, which was focused on delivering data center services using core Windows components.

The service that was publicly announced at the Microsoft Developer conference in 2008 was made up of five core services:

- Windows Azure: Allows users to spin up and manage compute instances, storage, and various networking services.
- **Microsoft SQL Data Services**: A cloud version of the Microsoft SQL database.
- **Microsoft .NET services**: Services that would allow you to deploy your .NET instance to the cloud without having to worry about instances.
- **Microsoft SharePoint and Microsoft Dynamics services**: These would be SaaS offerings of Microsoft's intranet and CRM software.

It launched in early 2010 to mixed reviews as some people thought it was limited compared to AWS, which by this time had been available for 4 years and was much more mature. However, Microsoft persevered, and over the last 10 years has added numerous services that have moved it way beyond its Windows roots. This promoted the name change from Windows Azure to Microsoft Azure in 2014.

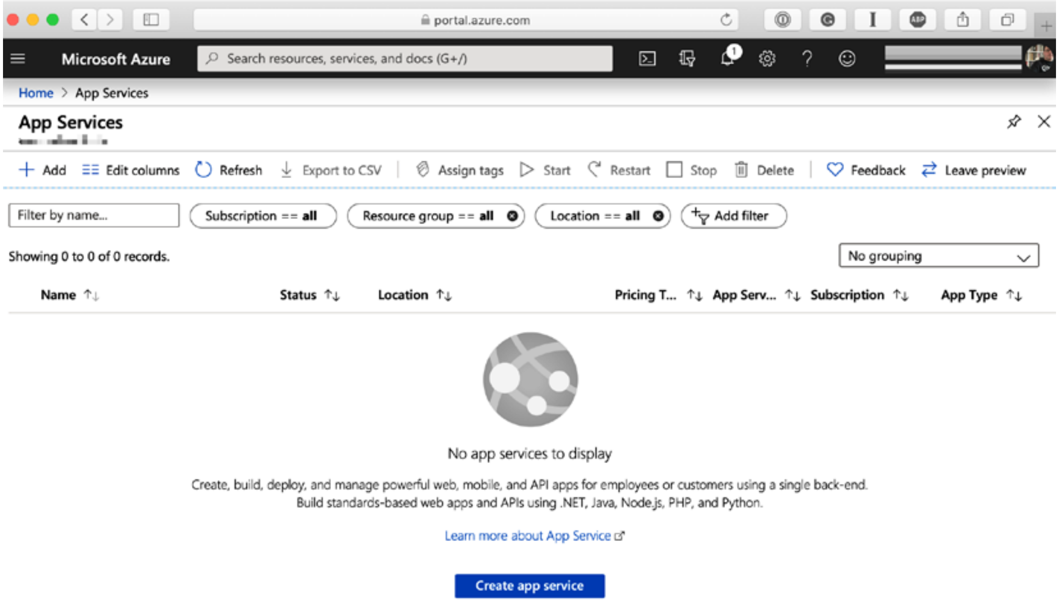
Since then, Microsoft's cloud has quickly caught up feature-wise with AWS, and depending on which news source you read, is chosen by enterprises to run their cloud workloads thanks to its tight integration with other Microsoft services, such as **Microsoft Office** and **Microsoft 365**.

Azure web app for containers

Microsoft Azure App Service is a fully managed platform that allows you to deploy your application and let Azure worry about managing the platform they are running on. There are several options available when launching an app service. You can run applications written in **.NET**, **.NET Core**, **Ruby**, **Node.js**, **PHP**, and **Python**, or you can launch an image directly from a container image registry.

In this quick walkthrough, we are going to be launching the cluster image from the Docker Hub:

1. Log in to the Azure portal at `https://portal.azure.com/`.
2. Select **App Services** from the left-hand menu, which can be accessed via the burger icon on the top left-hand corner of the screen:



Showing 0 to 0 of 0 records.

No app services to display

Create, build, deploy, and manage powerful web, mobile, and API apps for employees or customers using a single back-end. Build standards-based web apps and APIs using .NET, Java, Node.js, PHP, and Python.

Figure 10.12 – Preparing to launch an app service

3. On the page that loads, click on the **Create app service** button. You have several options to choose from here:

   Subscription: Choose a valid subscription.

   Resource Group: Click on **Create new** and follow the onscreen prompts.

   Name: Choose a unique name for the application.

   Publish: Select `Docker Container`.

   Operating System: Leave as **Linux**.

   Region: Select your preferred region.

   App Service plan: By default, a more expensive production-ready plan is selected, so clicking **Change size** in the **Sku and size** section will give you options on changing the pricing tier. For our needs, the **Dev/Test** plan will be fine.

4. Once you have selected and filled out the preceding options, click on the **Next Docker >** button. On this page, we have a few more bits of information to provide:

   Options: Leave this as `Single container`.

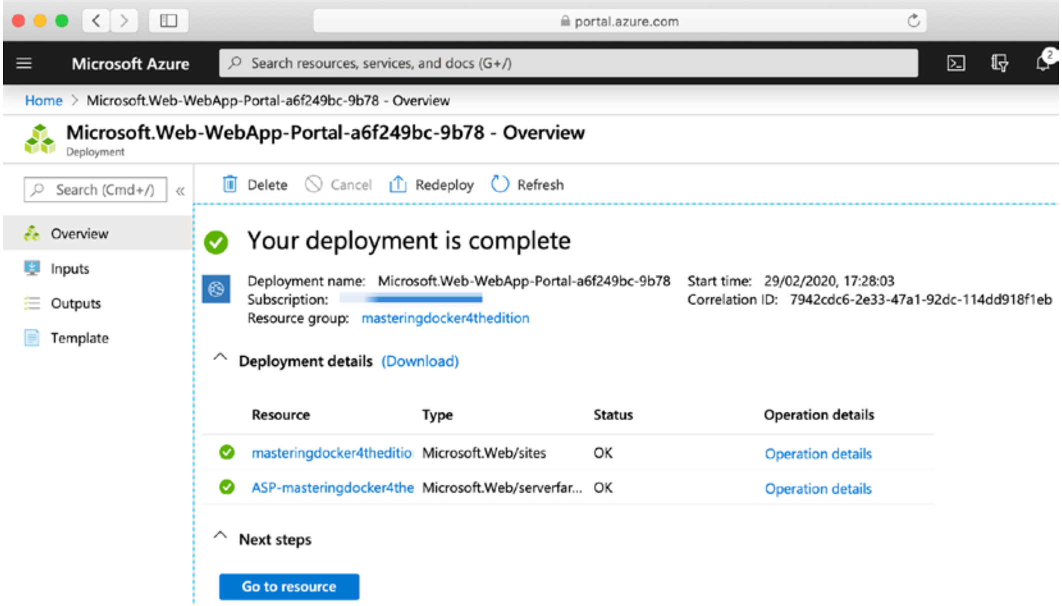
   Image source: Select `Docker Hub` from the dropdown list. This will open up the Docker Hub options below the form.

   Access type: Leave as `public`.

   Image and tag: Enter `russmckendrick/cluster:latest`.

   Startup command: Leave blank.

5. Once that's all completed, click through the **Monitoring** and **Tag** tabs, and then click on the **Create** button after reviewing the information. After a minute or two, you should be presented with a screen that looks like the following:



Microsoft.Web-WebApp-Portal-a6f249bc-9b78 - Overview

Deployment

Your deployment is complete

Deployment name: Microsoft.Web-WebApp-Portal-a6f249bc-9b78
Subscription: [redacted]
Resource group: masteringdocker4thedition

Start time: 29/02/2020, 17:28:03
Correlation ID: 7942cdc6-2e33-47a1-92dc-114dd918f1eb

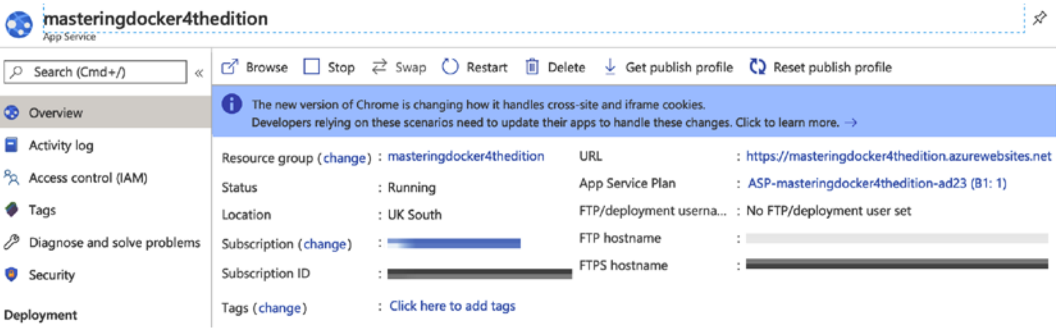
Deployment details (Download)

Resource	Type	Status	Operation details
masteringdocker4theditio	Microsoft.Web/sites	OK	Operation details
ASP-masteringdocker4the	Microsoft.Web/serverfar...	OK	Operation details

Next steps

Figure 10.13 – Deploying the app service

6. Clicking on the **Go to resource** button will take you to the newly launched application:



masteringdocker4thedition
App Service

Search (Cmd+/)

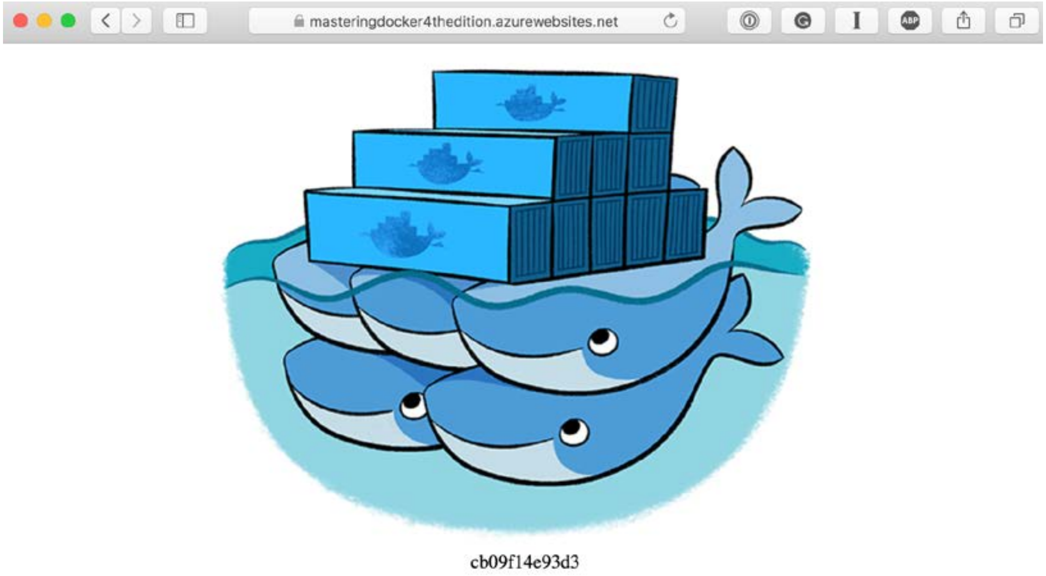
The new version of Chrome is changing how it handles cross-site and iframe cookies.
Developers relying on these scenarios need to update their apps to handle these changes. Click to learn more. →

Resource group (change) : masteringdocker4thedition
Status : Running
Location : UK South
Subscription (change) : [redacted]
Subscription ID : [redacted]
Tags (change) : Click here to add tags

URL : https://masteringdocker4thedition.azurewebsites.net
App Service Plan : ASP-masteringdocker4thedition-ad23 (B1: 1)
FTP/deployment userna... : No FTP/deployment user set
FTP hostname : [redacted]
FTPS hostname : [redacted]

Figure 10.14 – Viewing our running app service

Now that our application has launched, you should be able to access the service via the URL provided by Azure – for example, mine was `https://masteringdocker4thedition.azurewebsites.net`. Opening this, your browser will display the cluster application:



masteringdocker4thedition.azurewebsites.net

cb09f14e93d3

Figure 10.15 – Displaying the cluster application

As you can see, this time, we have the container ID, rather than a full hostname as we got when launching the container on AWS Fargate. The container at this spec will cost us around $10 per month.

There are some other really useful advantages to launching your container as an application, rather than just a plain old container – for example, you may have noticed that our container URL has an **SSL certificate** enabled. While it is currently one that covers `azurewebsites.net`, you can add your own custom domain and provide your own SSL certificate.

Another handy feature is that you can configure your single container to automatically update itself via a trigger from a webhook. For example, when your new container image has successfully been built, you can find this option on the **Container settings** page for the application:

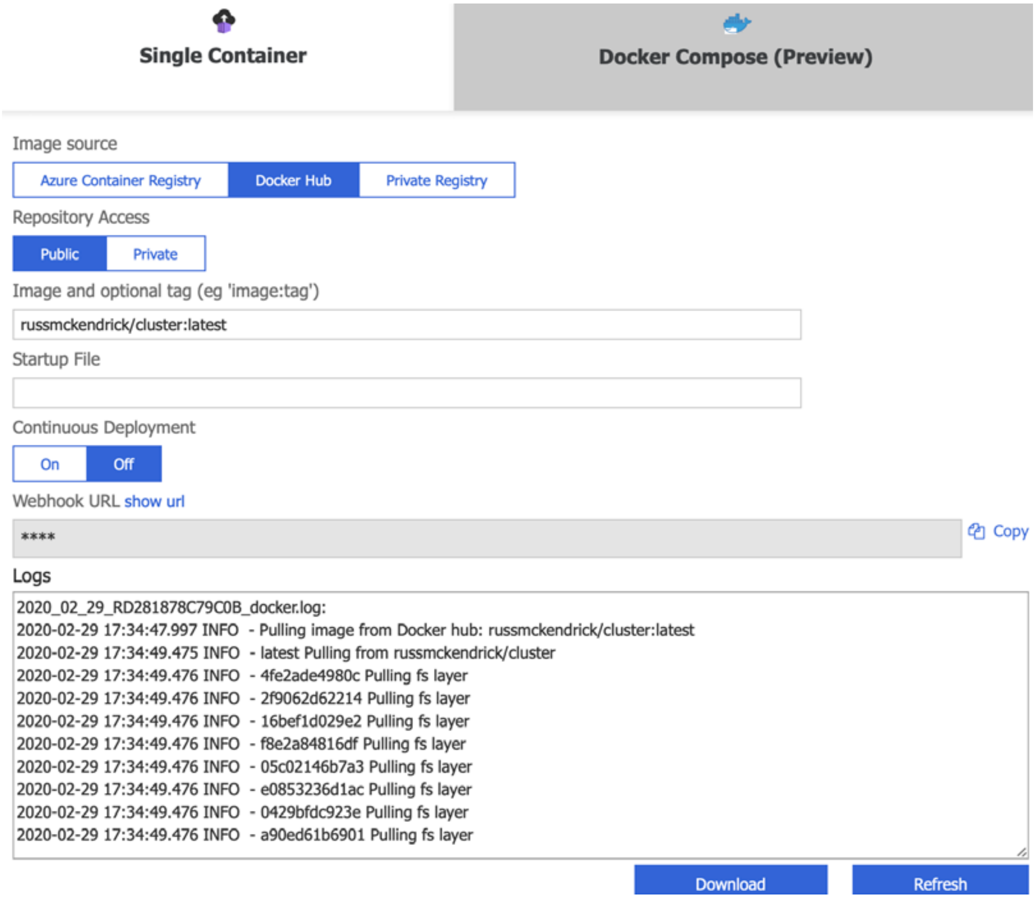


Image source: Docker Hub

Repository Access: Public

Image and optional tag (eg 'image:tag'): russmckendrick/cluster:latest

Startup File:

Continuous Deployment: Off

Webhook URL: ****

Logs

```
2020_02_29_RD281878C79C0B_docker.log:
2020-02-29 17:34:47.997 INFO  - Pulling image from Docker hub: russmckendrick/cluster:latest
2020-02-29 17:34:49.475 INFO  - latest Pulling from russmckendrick/cluster
2020-02-29 17:34:49.476 INFO  - 4fe2ade4980c Pulling fs layer
2020-02-29 17:34:49.476 INFO  - 2f9062d62214 Pulling fs layer
2020-02-29 17:34:49.476 INFO  - 16bef1d029e2 Pulling fs layer
2020-02-29 17:34:49.476 INFO  - f8e2a84816df Pulling fs layer
2020-02-29 17:34:49.476 INFO  - 05c02146b7a3 Pulling fs layer
2020-02-29 17:34:49.476 INFO  - e0853236d1ac Pulling fs layer
2020-02-29 17:34:49.476 INFO  - 0429bfdc923e Pulling fs layer
2020-02-29 17:34:49.476 INFO  - a90ed61b6901 Pulling fs layer
```

Figure 10.16 – Launching multiple containers

Also, as touched on when we first configured the application, you can use a Docker Compose file to launch multiple containers in your Microsoft Azure web app.

Once you have finished with the web app, delete it and the resource group, assuming it doesn't contain any other resources you need.

Azure container instances

Now that we have learned how to launch Docker containers in Azure using Azure web apps, let's now look at the Azure Container Instance service. Think of this as being similar in concept to the AWS Fargate service in that it allows you to launch containers directly on, in this case, Microsoft's shared backplane.

Let's configure a container instance:

1. Enter `Container instances` into the search bar at the top of the screen and then click on the link for the **Container Instances** service.
2. Once the page loads, click on the **Create container instances** button. This will take you to a page that looks quite like the one we saw when we launched the Azure web app:

   Subscription: Choose a valid subscription.

   Resource Group: Click on **Create new** and follow the onscreen prompts.

   Container Name: Choose a unique name for the application.

   Region: Select your preferred region.

   Image type: Leave this as `Public`.

   Image: Enter `russmckendrick/cluster:latest`.

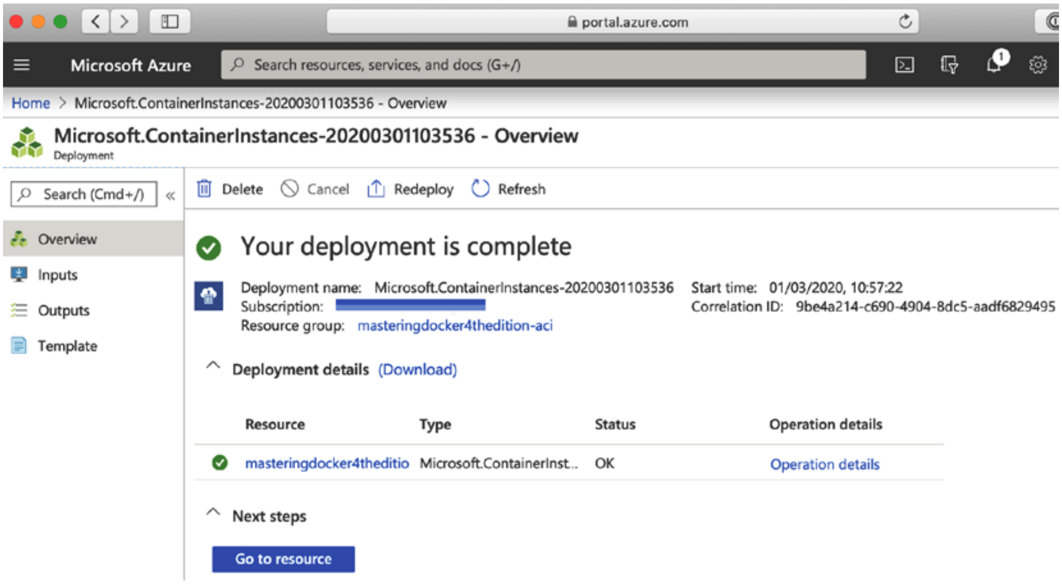
   OS Type: Leave as `Linux`.

   Size: I clicked on **Change size** and changed the memory to `1GB`.
3. Once you have filled in the information, click on the **Next: Networking >** button:

   Include public IP address: Leave as `Yes`.

   Ports: Leave as port `80` and **TCP** configured.

   DNS name label: Enter a DNS name for your container.
4. Skip past the **Advance** and **Tags** sections and go straight to **Review + create**. Once the validation has passed, click on the **Create** button:



Home > Microsoft.ContainerInstances-20200301103536 - Overview

Microsoft.ContainerInstances-20200301103536 - Overview
Deployment

Your deployment is complete

Deployment name: Microsoft.ContainerInstances-20200301103536
Subscription: [redacted]
Resource group: masteringdocker4thedition-aci
Start time: 01/03/2020, 10:57:22
Correlation ID: 9be4a214-c690-4904-8dc5-aadf6829495

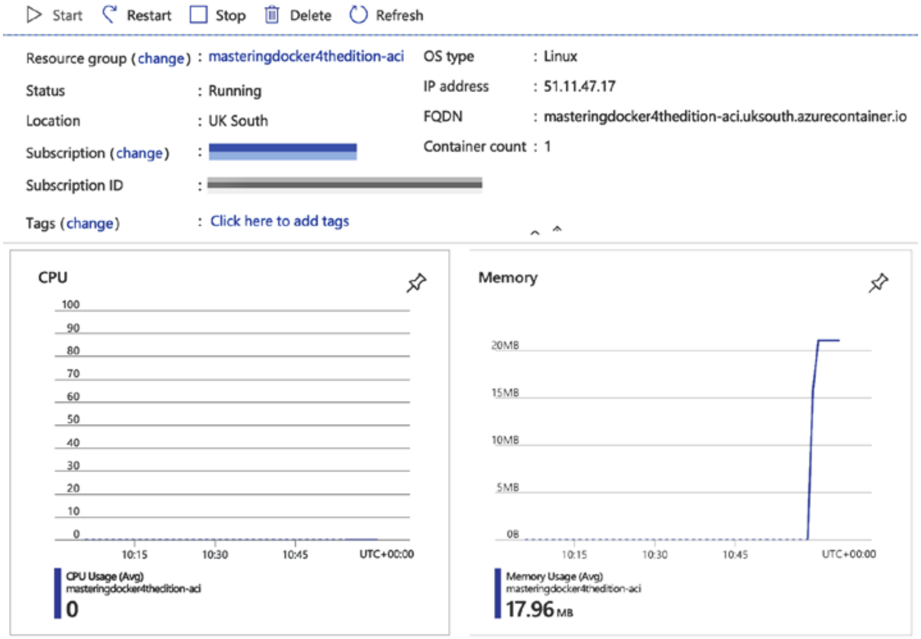
Deployment details (Download)

Resource	Type	Status	Operation details
masteringdocker4theditio	Microsoft.ContainerInst...	OK	Operation details

Next steps

Figure 10.17 – Launching our Azure container instance

5. As before, click on the **Go to resource** button and you will be taken to the newly created Azure container instance, as you can see in the following screenshot:



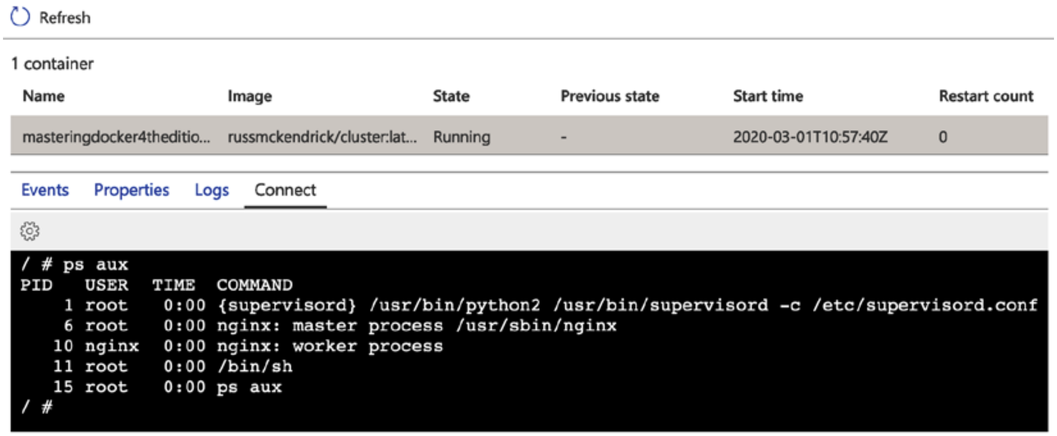
Resource group (change) : masteringdocker4thedition-aci
Status : Running
Location : UK South
Subscription (change) : [redacted]
Subscription ID : [redacted]
Tags (change) : Click here to add tags

OS type : Linux
IP address : 51.11.47.17
FQDN : masteringdocker4thedition-aci.uksouth.azurecontainer.io
Container count : 1

Figure 10.18 – An overview of our Azure container instance

There are fewer options than we had in the web application, and that is because Azure container instances are designed to do just one thing: run containers.

If you were to click on **Containers** in the left-hand menu, you would be able to enter the running container by clicking on the **Connect** tab and selecting which shell to connect to. In our case, as our container is based on Alpine Linux, we would need to choose the /bin/sh option:

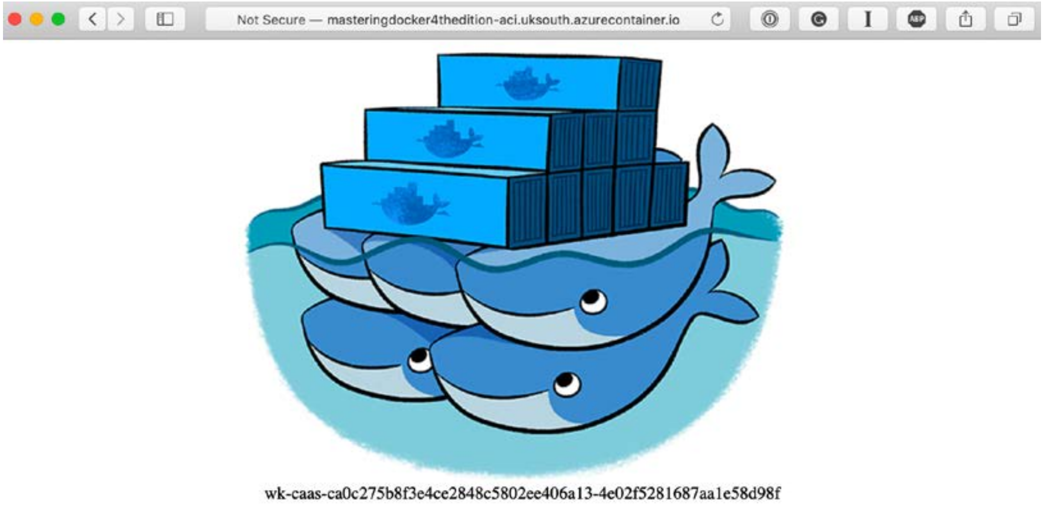


Name	Image	State	Previous state	Start time	Restart count
masteringdocker4theditio...	russmckendrick/cluster:lat...	Running	-	2020-03-01T10:57:40Z	0

```
/ # ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 {supervisord} /usr/bin/python2 /usr/bin/supervisord -c /etc/supervisord.conf
    6 root      0:00 nginx: master process /usr/sbin/nginx
   10 nginx     0:00 nginx: worker process
   11 root      0:00 /bin/sh
   15 root      0:00 ps aux
/ #
```

Figure 10.19 – Opening a session to our Azure container instance

Entering the URL given in the overview page into a browser, which in my case was `http://masteringdocker4thedition-aci.uksouth.azurecontainer.io/`, will show you your container app:


Not Secure — masteringdocker4thedition-aci.uksouth.azurecontainer.io

wk-caas-ca0c275b8f3e4ce2848c5802ee406a13-4e02f5281687aa1e58d98f

Figure 10.20 – Our running application

As you can see, this time we have the container ID, which in my case was wk-caas-ca0c275b8f3e4ce2848c5802ee406a13-4e02f5281687aa1e58d98f. You might notice that there is no HTTPS this time, just plain old HTTP on port 80.

Once you have finished with your Azure container instance, click on **Delete**, and also remove the resource group if required.

Summing up Microsoft Azure

While on the face of it the two services we have looked at in this section appear to quite similar, they are in fact really very different. Microsoft web apps are a managed service offered by Microsoft that is powered by containers. Typically, the container that is launched is for the code that the end user launches. However, when running containers, they end up running Docker in Docker. Azure container instances are just that, your running container – no wrappers or helpers, only vanilla containers.

We know have a foundation in Microsoft Azure. With Amazon and Microsoft established in the game, it's no surprise that Google launched its own competitor product. Let's take a look at it in the next section.

Google Cloud

Of the three major public clouds, **Google Cloud** is the newest. It started life as Google App Engine in 2008. App Engine was Google's PaaS offering, which supported Java, PHP, Node.js, Python, C#, .Net, Ruby, and Go applications. Unlike AWS and Microsoft Azure, Google remained a PaaS service for over 4 years until it introduced Google Compute Engine.

We are going to be learning a lot more about Google's journey into the cloud in the next chapter when we start to talk about Kubernetes, so I am not going to go into much more detail here. So, let's jump right in.

Google Cloud Run

Google Cloud Run works slightly differently than the other container services we have looked at in this chapter. The first thing we need is to have an image hosted in Google Container Registry to use the service:

1. Let's grab a copy of our cluster image from Docker Hub:

```
$ docker image pull russmckendrick/cluster
```

2. Now, we need to use the Google Cloud command-line tool to log in to our Google Cloud account. To do this, run the following:

```
$ gcloud init
```

3. Once logged in, we can configure Docker to use Google Container Registry by running the following:

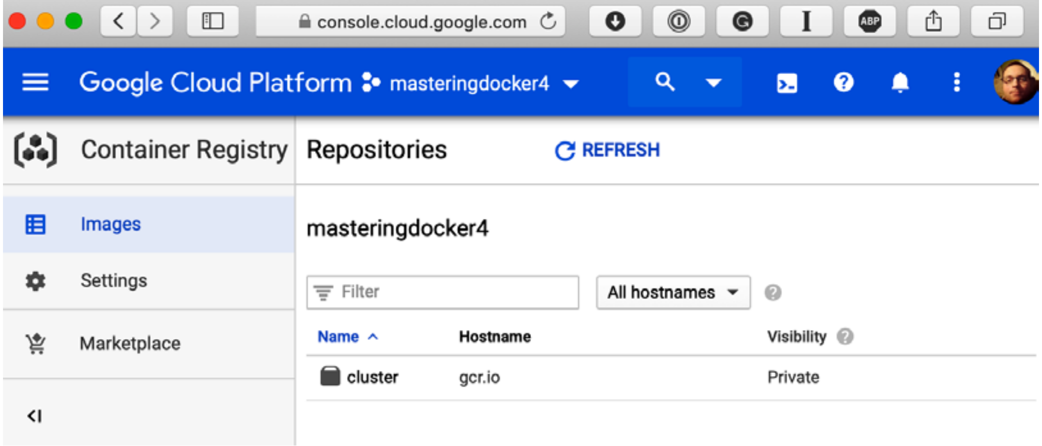
```
$ gcloud auth configure-docker
```

4. Now that Docker is configured to interact with Google Container Registry, we can run the following commands to create a tag and push our image:

```
$ docker image tag russmckendrick/cluster gcr.io/
masteringdocker4/cluster
```

```
$ docker image push gcr.io/masteringdocker4/
cluster:latest
```

You will notice that I have used a Google Cloud project name of `masterdocker4`; you will need to replace that with your own Google Cloud project name. You can verify that the image has been pushed by logging in to the Google Cloud console and navigating to Google Container Registry by entering `Cloud Registry` into the search bar at the top of the page. You should see something like the following:



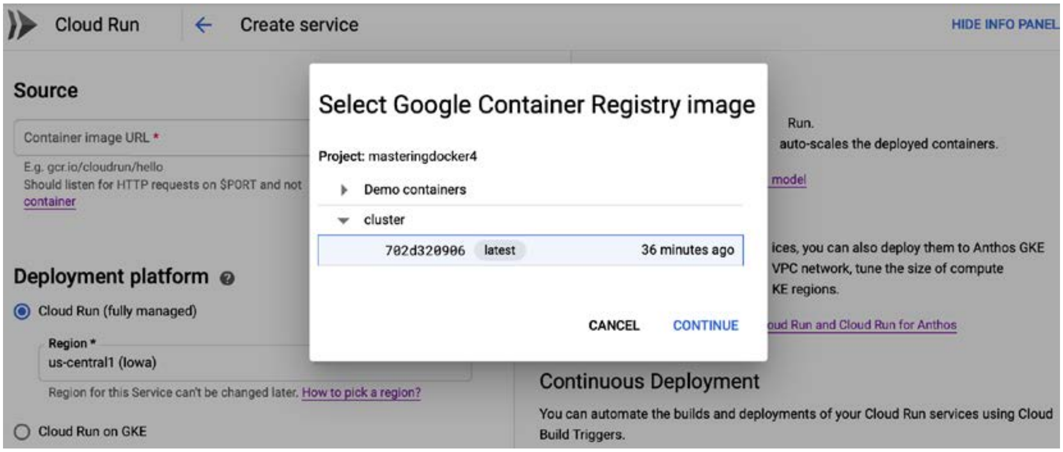
masteringdocker4

Filter | All hostnames

Name	Hostname	Visibility
cluster	gcr.io	Private

Figure 10.21 – Viewing our cluster image in Google Container Registry

5. Now, in the search bar at the top of the page, enter `Cloud Run` and follow the link.
6. Once the Cloud Run page loads, click on the **Create Service** button at the top.
7. The first thing you need to do is choose which image to use. Click on **Select** and highlight the cluster image and then the version of the image you want to use:



Container image URL *

E.g. gcr.io/cloudrun/hello
Should listen for HTTP requests on $PORT and not container

● Cloud Run (fully managed)

Region *
us-central1 (Iowa)

Region for this Service can't be changed later. How to pick a region?

○ Cloud Run on GKE

Select Google Container Registry image

Project: masteringdocker4

Demo containers

cluster

702d320906 latest 36 minutes ago

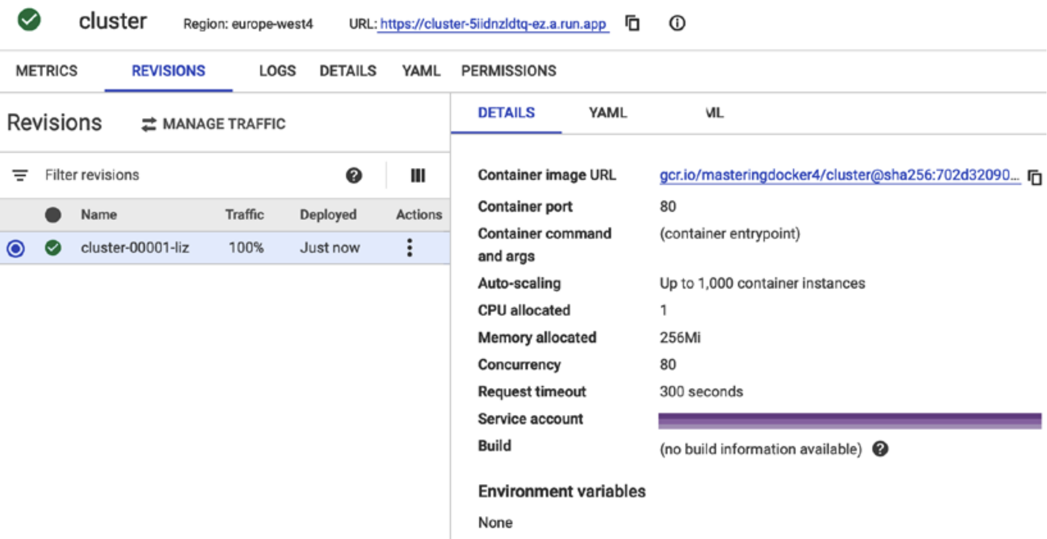
Run.
auto-scales the deployed containers.
model

ices, you can also deploy them to Anthos GKE VPC network, tune the size of compute KE regions.
oud Run and Cloud Run for Anthos

You can automate the builds and deployments of your Cloud Run services using Cloud Build Triggers.

Figure 10.22 – Selecting the Google Container Registry image

8. Once selected, click on **CONTINUE**.
9. For the deployment platform, we are going to be using **Cloud Run (fully managed)**. Choose the closest region to you from the drop-down box.
10. Next up, we need to name the server – I called mine `masteringdocker4cluster` – and then tick the **Allow unauthenticated invocations** radio box.
11. As our container listens on port `80`, we need to update the revision settings as the default is port `8080`. Clicking on **Revision Settings** will open up more options, the first of which is for the container port. Change this from `8080` to `80`.
12. Once everything is filled in, scroll to the bottom of the page and click on the **Create** button. After a short while, you should see something like the following screen:



cluster Region: europe-west4 URL: https://cluster-5iidnzldtq-ez.a.run.app

Revisions

	●	Name	Traffic	Deployed	Actions
◉	✓	cluster-00001-liz	100%	Just now	⋮

Container image URL	gcr.io/masteringdocker4/cluster@sha256:702d32090...
Container port	80
Container command and args	(container entrypoint)
Auto-scaling	Up to 1,000 container instances
CPU allocated	1
Memory allocated	256Mi
Concurrency	80
Request timeout	300 seconds
Service account	[redacted]
Build	(no build information available)

Environment variables

None

Figure 10.23 – Viewing our Cloud Run application

As you can see, we have information on our running container, including a URL, which in this case was `https://cluster-5iidnzldtq-ez.a.run.app` – opening the URL in a browser shows the following:

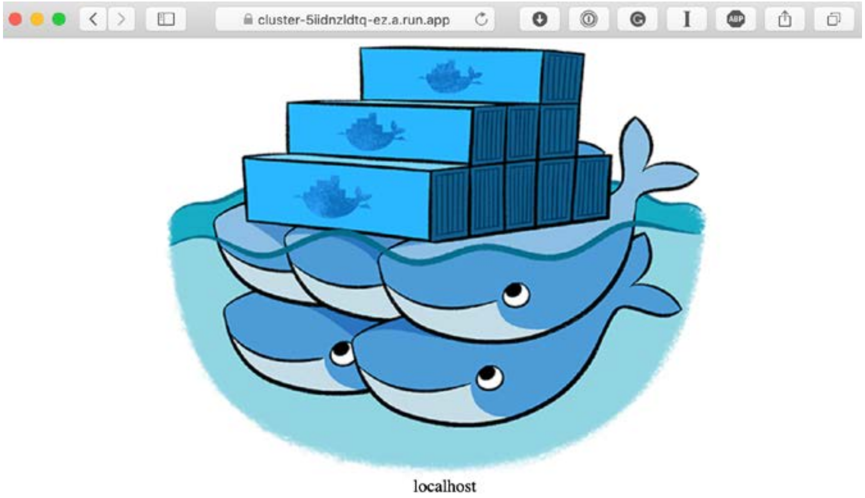


Figure 10.24 – Our running application

This is what we would have expected to have seen; however, there is a difference between how Google Cloud Run and the other services we covered in the AWS and Azure section of this chapter work. With the other service, our container was running all of the time, but with Google Cloud Run, it only runs when it is needed. Google Cloud Run is built on top of **Knative**, which is an open source serverless platform designed to run on top of a Kubernetes, and we have been running our containers on Google's own Kubernetes cluster.

Summing up Google Cloud

As you may have guessed, Google is more geared to running Kubernetes-based services, as we have already seen with Google Cloud Run. So, we will revisit Google Cloud in a later chapter once we have covered Kubernetes in a little more depth.

Summary

In this chapter, we have looked at how we can deploy our Docker containers into services offered by the three leading public cloud providers using their container-only services. The services we looked at all approach both the deployment and management of containers in very different ways, from the fully managed Docker-based web app in Azure app services to AWS' own clustering service Amazon ECS.

The differences between all of these services are quite important as it means that if you want to use them, then you are tied to just the one cloud provider. While in most cases, that shouldn't be too much of a problem, it could in the long term end up limiting you.

In the next chapter (and in subsequent chapters), we will explore one of the most exciting services to arrive on the scene since Docker: Kubernetes.

Questions

1. What type of application do we need to launch in Azure?
2. What Amazon service don't you have to manage if you're using Amazon Fargate directly?
3. True or false: Azure Container Instances comes with HTTPS support out of the box.
4. Name the open source service that Google Cloud Run is built on top of.
5. Which of the services that we looked at support Docker Compose?

Further reading

For details on each of the services we have used in this chapter, refer to the following:

- AWS: `http://aws.amazon.com/`
- Amazon ECS: `https://aws.amazon.com/ecs/`
- AWS Fargate: `https://aws.amazon.com/fargate/`
- Microsoft `Azure: https://azure.microsoft.com/`
- Azure App Service: `https://azure.microsoft.com/en-gb/services/app-service/`
- Azure Container Instances: `https://azure.microsoft.com/en-gb/services/container-instances/`
- Google Cloud: `https://cloud.google.com`
- Google Cloud Run: `https://cloud.google.com/run`
- Knative: `http://knative.dev`

11
Docker and Kubernetes

In this chapter, we will be taking a look at Kubernetes. Like Docker Swarm, you can use Kubernetes to create and manage clusters that run your container-based applications.

We will be discussing the following topics in this chapter:

- An introduction to Kubernetes
- Enabling Kubernetes in Docker Desktop
- Using Kubernetes and Docker Desktop
- Kubernetes and other Docker tools

Technical requirements

Kubernetes within Docker only supports by Docker for Mac and Docker for Windows desktop clients. If you are running Linux then in the next chapter, *Chapter 12, Discovering more Kubernetes options*, we are going to be looking at some options that will be | relevant to you.

Like previous chapters, I will be using my preferred operating system, which is macOS. As before, some of the supporting commands, which will be few and far between, may only apply to macOS.

Check out the following video to see the Code in Action:
`https://bit.ly/3m1WRiw`

An introduction to Kubernetes

If you have been thinking about looking at containers, then you would have come across Kubernetes at some point on your travels, so before we enable it within our Docker desktop installation, let's take a moment to look at how Kubernetes started life.

Kubernetes, pronounced **koo-ber-net-eez**, originates from the Greek name given to a helmsman or captain of a ship.

> **Info**
>
> The method of shortening the name adopted by the Kubernetes team is called a numeronym and was devised in the 80s, and is still used today. See `https://en.wikipedia.org/wiki/Numeronym` for more information.

Kubernetes, which is also known as **K8s** – the number 8 in the K8s shorthand represents the number of letters between the K and S, the 'ubernete' part – is an open source project that originated at Google and allows you to automate the deployment, management, and scaling of your containerized applications.

A brief history of containers at Google

Google has been working on Linux container-based solutions for quite a long time. It took its first steps in 2006 by working on the Linux kernel feature called **Control Groups** (**cgroups**). This feature was merged into the Linux kernel in 2008 within release 2.6.24.

The feature allows you to isolate resources, such as CPU, RAM, networking, and disk I/O, or one or more processes. Control Groups remains a core requirement for Linux containers and is not only used by Docker but also other container tools.

Google next dipped their toes into the container waters with a container stack called **lmctfy**, which stands for **Let Me Contain That For You** and was an alternative to the **LXC** collection of tools and libraries. It was an open sourced version of Google's internal toolset, which they used to manage containers in their applications.

The next time Google hit the news about their container usage was following a talk given by *Joe Beda* at *Gluecon* in May 2014. During the presentation, Beda revealed that pretty much everything within Google was container-based and that they were launching around 2 billion containers a week. It was stated that this number did not include any long-running containers, meaning that the containers were only active for a short amount of time. However, after some quick math, on average Google was launching around 3,000 containers per second!

Later in the talk, Beda mentioned that Google was using a scheduler, so they didn't have to manually manage 2 billion containers a week or even worry about where they were launched and, to a lesser extent, each container's availability.

Google also published a paper called *Large-scale cluster management at Google with Borg*. This paper not only let people outside of Google know the name of the scheduler they were using, **Borg**, but it also went into great detail about the design decisions they made when designing the scheduler.

The paper mentioned that as well as their internal tools, Google was running its customer-facing applications, such as Google Docs, Google Mail, and Google Search, in containers running clusters, which are managed by Borg.

Borg was named after the alien race, the Borg, from the Star Trek: The Next Generation TV show. In the TV show, the Borg are a race of cybernetic beings whose civilization is founded on a hive mind known as the collective. This gives them not only the ability to share the same thoughts but also, through a sub-space network, ensure that each member of the collective is given guidance and supervision from the collective consciousness. I am sure you will agree, the characteristics of the Borg race matches that closely how you would want your cluster of containers to run.

Borg was running within Google for several years and it was eventually replaced by a more modern scheduler called Omega. It was around this time that Google announced it that it would be taking some of the core functionality of Borg and reproducing it as a new open source project. This project, known internally as **Seven**, was worked on by several of the core contributors to Borg. It aimed to create a friendlier version of Borg that wasn't closely tied into Google's own internal procedures and ways of working.

Seven, named after the *Star Trek: Voyager character* Seven of Nine, who was a Borg that broke away from the collective, would eventually be named **Kubernetes** by the time of its first public commit.

So, now that we know how Kubernetes came to be, we can dig a little deeper into what Kubernetes is.

An overview of Kubernetes

The bulk of the project, 90.7% at the time of writing this, is written in Go, which should come as no surprise as Go is a programming language that was developed internally at Google before it was open sourced in 2011. The rest of the project files are made up of Python and Shell helper scripts and HTML documentation.

A typical Kubernetes cluster is made up of servers that take on either a master or node role. You can also run a standalone installation that takes on both roles.

The master role is where the magic happens, and it is the brains of the cluster. It is responsible for making decisions on where pods are launched and for monitoring the health of both the cluster itself and also of the pods running within the cluster. We will discuss pods once we have finished looking at the two roles.

Typically, the core components that are deployed to a host that has been given the role of a master are as follows:

- `kube-apiserver`: This component exposes the main Kubernetes API. It is designed to horizontally scale, which means that you can keep adding more instances of it to make your cluster highly available.
- `etcd`: This is a highly available consistent key-value store. It is used to store the state of the cluster.
- `kube-scheduler`: This component is responsible for making the decisions on where pods are launched.
- `kube-controller-manager`: This component runs controllers. These controllers have several functions within Kubernetes, such as monitoring the nodes, keeping an eye on the replication, managing the endpoints, and generating service accounts and tokens.
- `cloud-controller-manager`: This component takes on the management of the various controllers, which interact with third-party clouds to launch and configure supporting services.

Now that we have our management components covered, we need to discuss what they are managing. A node is made up of the following elements:

- `kubelet`: This agent runs on each node within the cluster, and it is the means by which the managers interact with the nodes. It is also responsible for managing the pods.
- `kube-proxy`: This component manages all of the routing of requests and traffic for both the node and also the pods.

- `container runtime`: This could be Docker, CRI-O, or any other OCI-compliant runtime.

You may have noticed that I have not mentioned containers much so far. This is because Kubernetes doesn't actually directly interact with your containers; instead, it communicates with a pod. Think of a pod as a complete application, a little like when we looked at launching an application made up of multiple containers using Docker Compose.

How does Docker fit in with Kubernetes?

Docker's relationship with Kubernetes is varied. To start with, Docker, the container engine, powers a lot of Kubernetes installations in one form or another, for example, as Docker or ContainerD.

However, Kubernetes was originally seen as a competitive technology to Docker Swarm, which was Docker's own clustering technology. However, over the last few years, Kubernetes has emerged as pretty much the de facto standard for container clustering/orchestration.

All of the major cloud providers provide Kubernetes-as-a-service. We have the following:

- **Google Cloud**: **Google Kubernetes Engine** (**GKE**)
- **Microsoft Azure**: **Azure Kubernetes Service** (**AKS**)
- **Amazon Web Services**: **Amazon Elastic Container Service for Kubernetes** (**EKS**)
- **IBM**: **IBM Cloud Kubernetes Service**
- **Oracle Cloud**: **Oracle Container Engine for Kubernetes**
- **DigitalOcean**: **Kubernetes on DigitalOcean**

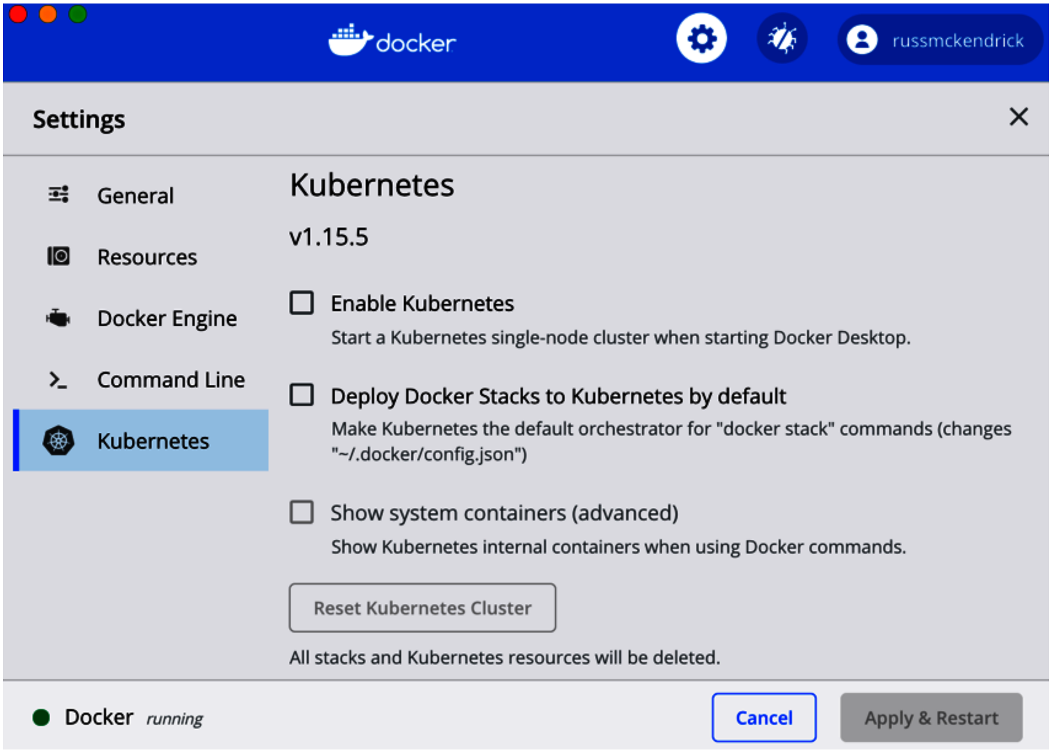
On the face of it, all of the major players supporting Kubernetes may not seem like that big a deal. However, consider that we now know a consistent way of deploying our containerized applications across multiple platforms. Traditionally, these platforms have been walled gardens and have very different ways of interacting with them.

While Docker's announcement of Kubernetes support in its desktop versions in October 2017 at DockerCon Europe initially came as a surprise, once the dust settled the announcement made perfect sense. Providing developers with an environment where they could work on their applications locally using Docker for Mac and Docker for Windows, and then using Docker Enterprise Edition to deploy and manage their own Kubernetes clusters, or even use one of the cloud services mentioned previously, fits in with trying to solve the 'works on my machine' problem we discussed in *Chapter 1*, *Docker Overview*.

Let's now take a look at how you can enable support in the Docker software and get stuck in using it.

Enabling Kubernetes in Docker Desktop

Docker has made the installation process extremely simple. All you need to do to enable Kubernetes support is open **Preferences** and click on the **Kubernetes** tab:



Kubernetes

v1.15.5

□ Enable Kubernetes

Start a Kubernetes single-node cluster when starting Docker Desktop.

□ Deploy Docker Stacks to Kubernetes by default

Make Kubernetes the default orchestrator for "docker stack" commands (changes "~/.docker/config.json")

□ Show system containers (advanced)

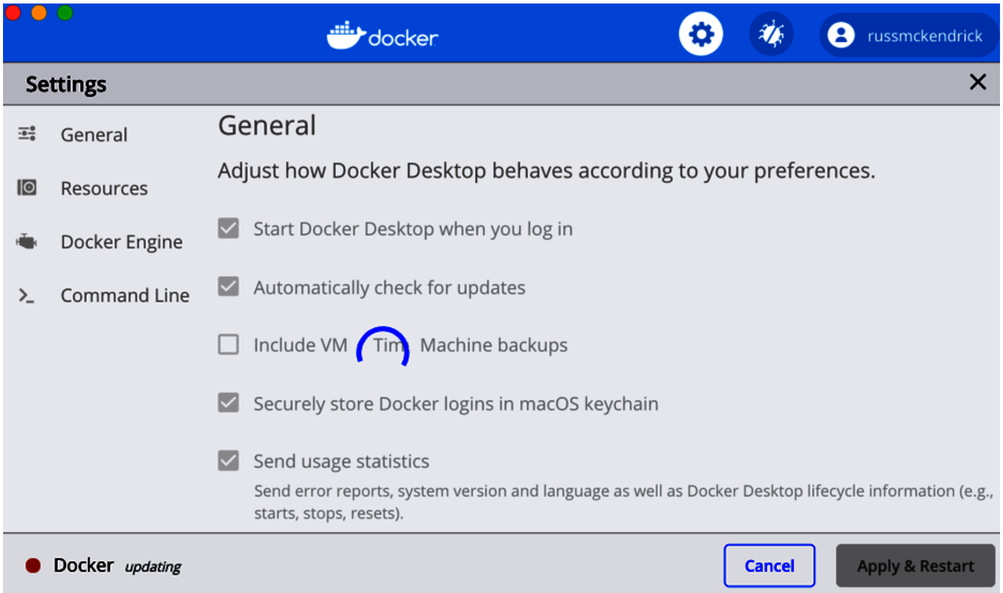
Show Kubernetes internal containers when using Docker commands.

All stacks and Kubernetes resources will be deleted.

Figure 11.1 – The Kubernetes preferences in Docker for Mac

As you can see, there are three options. Tick the **Enable Kubernetes** box and then select **Deploy Docker Stacks to Kubernetes by default**. Leave **Show systems containers (advanced)** unticked for now; we look at this in a little more detail once we have enabled the service.

Clicking **Apply & Restart** will do just that, restart Docker and enable Kubernetes:



General

Adjust how Docker Desktop behaves according to your preferences.

☑ Start Docker Desktop when you log in

☑ Automatically check for updates

□ Include VM Time Machine backups

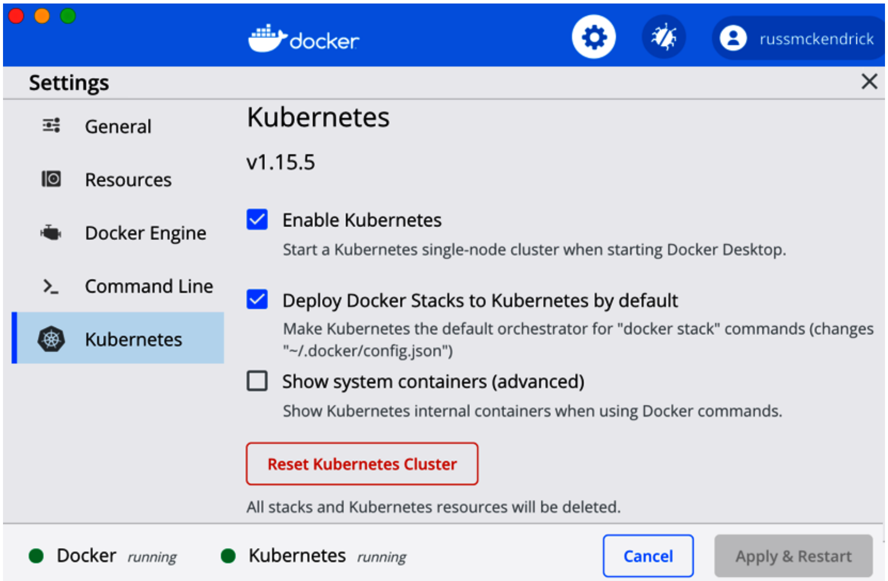
☑ Securely store Docker logins in macOS keychain

☑ Send usage statistics
Send error reports, system version and language as well as Docker Desktop lifecycle information (e.g., starts, stops, resets).

● Docker *updating*

Figure 11.2 – Enabling Kubernetes on Docker for Mac

It will take a short while for Docker to download, configure, and launch the cluster. Once complete, you should see Docker and Kubernetes listed in the bottom left of the settings window. Both should have a green dot next to them to indicate that the services are running:



Kubernetes

v1.15.5

☑ Enable Kubernetes
Start a Kubernetes single-node cluster when starting Docker Desktop.

☑ Deploy Docker Stacks to Kubernetes by default
Make Kubernetes the default orchestrator for "docker stack" commands (changes "~/.docker/config.json")

□ Show system containers (advanced)
Show Kubernetes internal containers when using Docker commands.

All stacks and Kubernetes resources will be deleted.

● Docker *running* ● Kubernetes *running*

Figure 11.3 – Kubernetes successfully enabled on Docker for Mac

Open Terminal and run the following command:

```
$ docker container ls -a
```

This should show you that there is nothing out of the ordinary running. Run the following command:

```
$ docker image ls
```

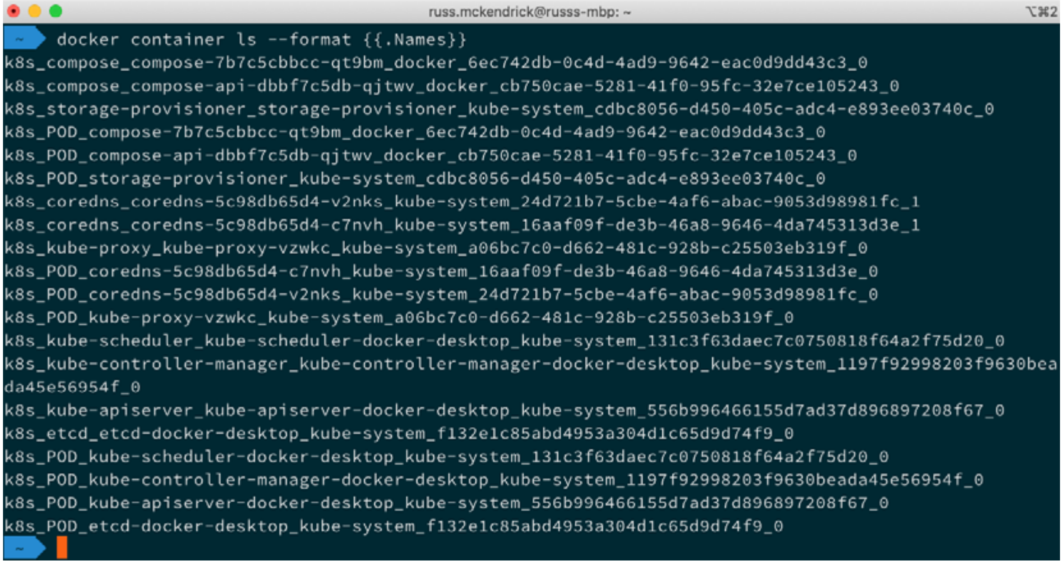
Again, this shows nothing of interest; however, as you might have guessed, ticking the **Show system containers (advanced)** option in the **Settings** window will change this. Tick it now and then re-run the following command:

```
$ docker container ls -a
```

As there is a lot of output when running the preceding command, the following screenshot shows just the names of the containers. To do this, I ran the following:

```
$ docker container ls --format {{.Names}}
```

Running the command gave me the following:



```
~  docker container ls --format {{.Names}}
k8s_compose_compose-7b7c5cbbcc-qt9bm_docker_6ec742db-0c4d-4ad9-9642-eac0d9dd43c3_0
k8s_compose_compose-api-dbbf7c5db-qjtwv_docker_cb750cae-5281-41f0-95fc-32e7ce105243_0
k8s_storage-provisioner_storage-provisioner_kube-system_cdbc8056-d450-405c-adc4-e893ee03740c_0
k8s_POD_compose-7b7c5cbbcc-qt9bm_docker_6ec742db-0c4d-4ad9-9642-eac0d9dd43c3_0
k8s_POD_compose-api-dbbf7c5db-qjtwv_docker_cb750cae-5281-41f0-95fc-32e7ce105243_0
k8s_POD_storage-provisioner_kube-system_cdbc8056-d450-405c-adc4-e893ee03740c_0
k8s_coredns_coredns-5c98db65d4-v2nks_kube-system_24d721b7-5cbe-4af6-abac-9053d98981fc_1
k8s_coredns_coredns-5c98db65d4-c7nvh_kube-system_16aaf09f-de3b-46a8-9646-4da745313d3e_1
k8s_kube-proxy_kube-proxy-vzwkc_kube-system_a06bc7c0-d662-481c-928b-c25503eb319f_0
k8s_POD_coredns-5c98db65d4-c7nvh_kube-system_16aaf09f-de3b-46a8-9646-4da745313d3e_0
k8s_POD_coredns-5c98db65d4-v2nks_kube-system_24d721b7-5cbe-4af6-abac-9053d98981fc_0
k8s_POD_kube-proxy-vzwkc_kube-system_a06bc7c0-d662-481c-928b-c25503eb319f_0
k8s_kube-scheduler_kube-scheduler-docker-desktop_kube-system_131c3f63daec7c0750818f64a2f75d20_0
k8s_kube-controller-manager_kube-controller-manager-docker-desktop_kube-system_1197f92998203f9630bea
da45e56954f_0
k8s_kube-apiserver_kube-apiserver-docker-desktop_kube-system_556b996466155d7ad37d896897208f67_0
k8s_etcd_etcd-docker-desktop_kube-system_f132e1c85abd4953a304d1c65d9d74f9_0
k8s_POD_kube-scheduler-docker-desktop_kube-system_131c3f63daec7c0750818f64a2f75d20_0
k8s_POD_kube-controller-manager-docker-desktop_kube-system_1197f92998203f9630beada45e56954f_0
k8s_POD_kube-apiserver-docker-desktop_kube-system_556b996466155d7ad37d896897208f67_0
k8s_POD_etcd-docker-desktop_kube-system_f132e1c85abd4953a304d1c65d9d74f9_0
~
```

Figure 11.4 – Listing the containers that make up our Kubernetes installation

There are 20 running containers, which is why you have the option of hiding them. As you can see, nearly all of the components we discussed in the previous section are covered as well as a few additional components, which provide the integration with Docker.

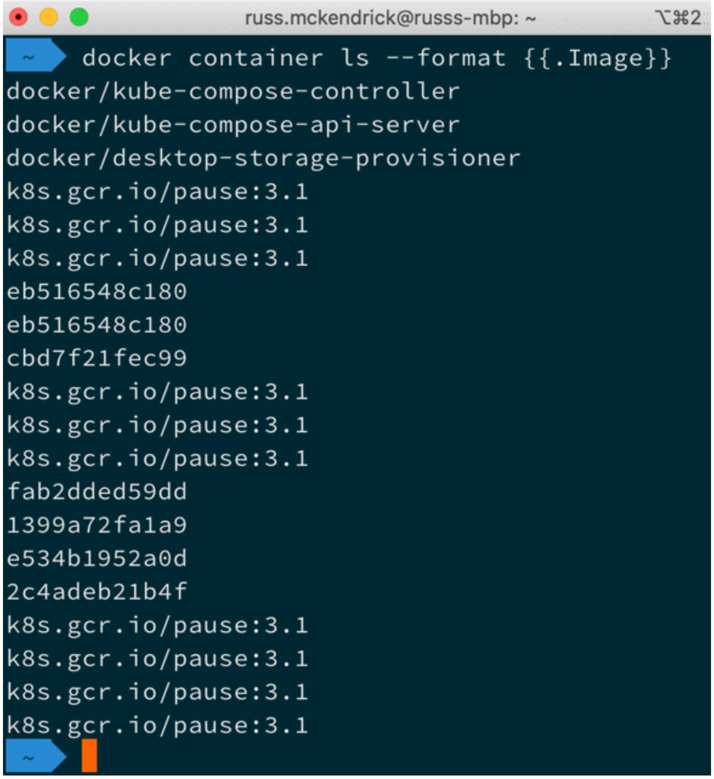
Run the following command:

```
$ docker image ls
```

It still doesn't list any images, although we get a list of images that are being used by running the following:

```
$ docker container ls --format {{.Image}}
```

As you can see from the following output, the images are sourced from both Docker and also the official Kubernetes images that are available from the Google Container Registry (k8s.gcr.io), and there are also some images that have been built locally:



```
~ docker container ls --format {{.Image}}
docker/kube-compose-controller
docker/kube-compose-api-server
docker/desktop-storage-provisioner
k8s.gcr.io/pause:3.1
k8s.gcr.io/pause:3.1
k8s.gcr.io/pause:3.1
eb516548c180
eb516548c180
cbd7f21fec99
k8s.gcr.io/pause:3.1
k8s.gcr.io/pause:3.1
k8s.gcr.io/pause:3.1
fab2dded59dd
1399a72fa1a9
e534b1952a0d
2c4adeb21b4f
k8s.gcr.io/pause:3.1
k8s.gcr.io/pause:3.1
k8s.gcr.io/pause:3.1
k8s.gcr.io/pause:3.1
~
```

Figure 11.5 – Viewing the images being used to power the Kubernetes installation

For now, I would recommend unticking the **Show system containers (advanced)** option, as we do not need to see a list of 20 containers running each time that we look at the running containers.

The other thing to note at this point is that the **Kubernetes** menu item in the Docker app now has content in it. This menu can be used for switching between Kubernetes clusters. As we only have one cluster active at the moment, there is only one listed:

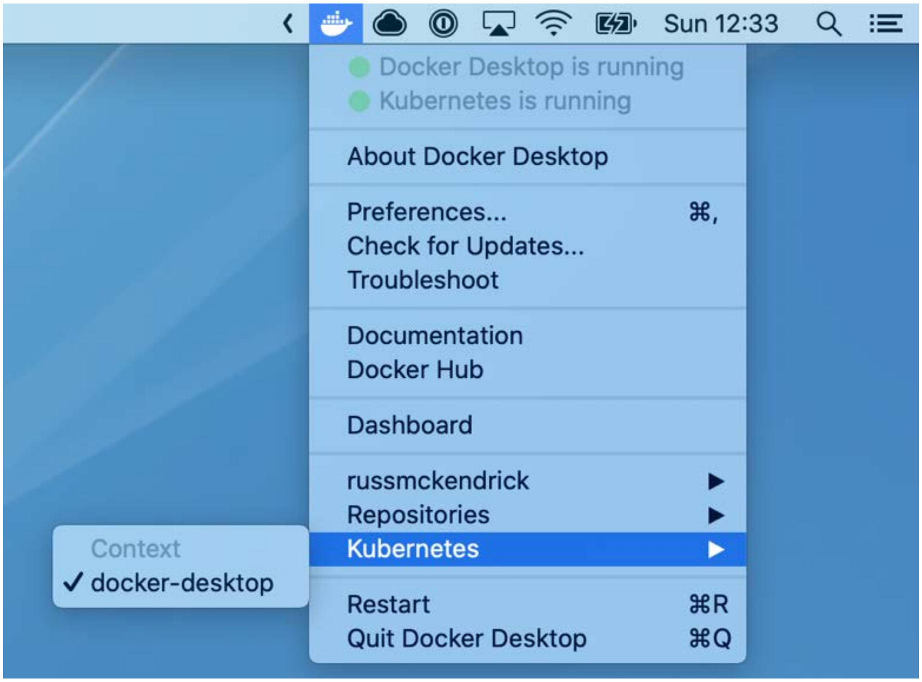


Figure 11.6 – Checking the Kubernetes menu item

Now that we have our local Kubernetes cluster up and running, we can start to use it.

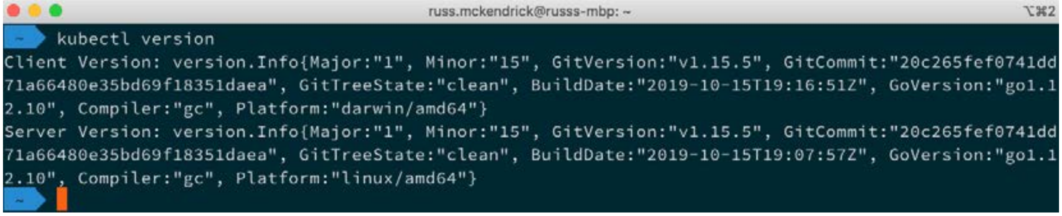
Using Kubernetes and Docker Desktop

Now that we have our Kubernetes cluster up and running on our Docker desktop installation, we can start to interact with it. To start with, we are going to look at the command line that was installed alongside the Docker desktop component, `kubectl`.

As mentioned, `kubectl` was installed alongside Docker. The following command will show some information about the client and also the cluster it is connected to:

```
$ kubectl version
```

Like when running `docker version`, this should give you information on both the client and server:



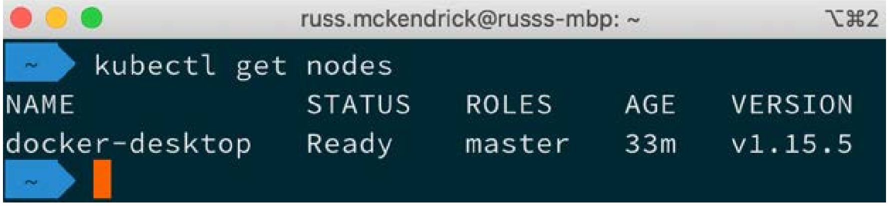
```
~ kubectl version
Client Version: version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.5", GitCommit:"20c265fef0741dd
71a66480e35bd69f18351daea", GitTreeState:"clean", BuildDate:"2019-10-15T19:16:51Z", GoVersion:"go1.1
2.10", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.5", GitCommit:"20c265fef0741dd
71a66480e35bd69f18351daea", GitTreeState:"clean", BuildDate:"2019-10-15T19:07:57Z", GoVersion:"go1.1
2.10", Compiler:"gc", Platform:"linux/amd64"}
~
```

Figure 11.7 – Checking the versions of the client and server

Next, we can run the following to see if `kubectl` can see our node:

```
$ kubectl get nodes
```

As we only have a single node, we should only see one listed:



```
~ kubectl get nodes
NAME             STATUS   ROLES    AGE   VERSION
docker-desktop   Ready    master   33m   v1.15.5
~
```

Figure 11.8 – Listing our nodes

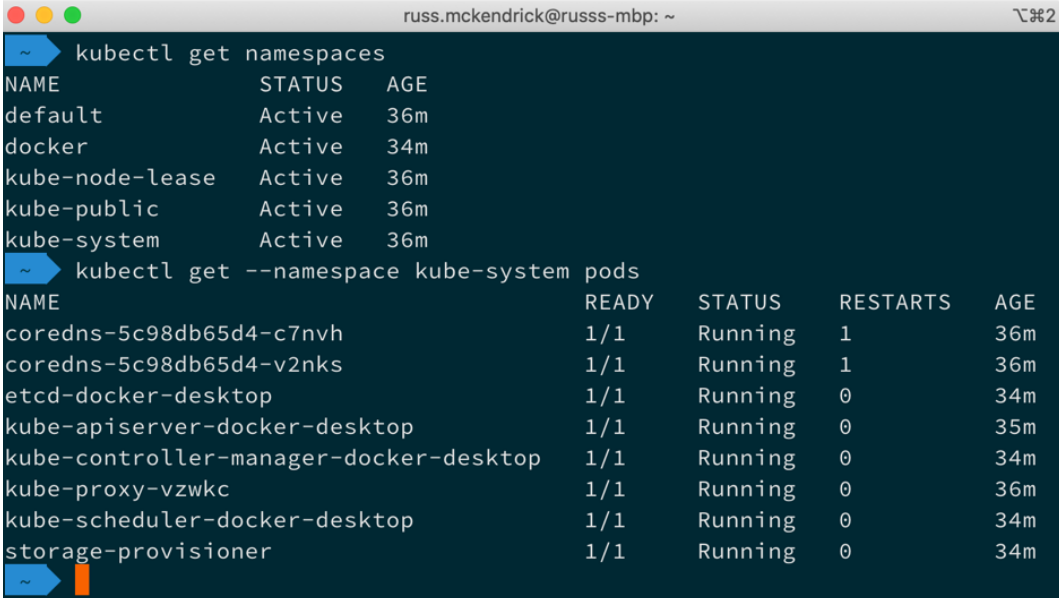
Now that we have our client interacting with our node, we can view the namespaces that are configured by default within Kubernetes by running the following command:

```
$ kubectl get namespaces
```

Then we can view the pods within a namespace with the following command:

```
$ kubectl get pods --namespace kube-system
```

What follows is the Terminal output I received when I ran the preceding commands:



russ.mckendrick@russs-mbp: ~

```
~ kubectl get namespaces
NAME              STATUS   AGE
default           Active   36m
docker            Active   34m
kube-node-lease   Active   36m
kube-public       Active   36m
kube-system       Active   36m
~ kubectl get --namespace kube-system pods
NAME                                     READY   STATUS    RESTARTS   AGE
coredns-5c98db65d4-c7nvh                 1/1     Running   1          36m
coredns-5c98db65d4-v2nks                 1/1     Running   1          36m
etcd-docker-desktop                      1/1     Running   0          34m
kube-apiserver-docker-desktop            1/1     Running   0          35m
kube-controller-manager-docker-desktop   1/1     Running   0          34m
kube-proxy-vzwkc                         1/1     Running   0          36m
kube-scheduler-docker-desktop            1/1     Running   0          34m
storage-provisioner                      1/1     Running   0          34m
~
```

Figure 11.9 – Checking the namespaces

Namespaces within Kubernetes are a great way of isolating resources within your cluster. As you can see from the Terminal output, there are four namespaces within our cluster. There is the `default` namespace, which is typically empty. There are two namespaces for the main Kubernetes services: `docker` and `kube-system`. These contain the pods that make up our cluster and the final namespace, `kube-public`, like the default namespace, is empty.

Before we launch our own pod, let's take a quick look at how we can interact with the pods we have running, starting with how we can find more information about our pod:

```
$ kubectl describe pods kube-scheduler-docker-desktop
--namespace kube-system
```

The preceding command will print out the details of the kube-scheduler-docker-desktop pod. You might notice that we had to pass the namespace using the --namespace flag. If we didn't, then kubectl would default to the default namespace where there isn't a pod called kube-scheduler-docker-desktop running.

The full output of the command is shown here, starting with some basic information on the pod:

```
Name:                 kube-scheduler-docker-desktop
Namespace:            kube-system
Priority:             2000000000
Priority Class Name:  system-cluster-critical
Node:                 docker-desktop/192.168.65.3
Start Time:           Sun, 03 May 2020 12:11:02 +0100
```

Like Docker, you can apply labels to pods. This is shown in the following screenshot, along with some more details around the pod:

```
Labels:               component=kube-scheduler
                      tier=control-plane
Annotations:          kubernetes.io/config.hash: 131c3f63daec7c
0750818f64a2f75d20
                      kubernetes.io/config.mirror: 131c3f63daec
7c0750818f64a2f75d20
                      kubernetes.io/config.seen:
2020-05-03T11:10:56.315367593Z
                      kubernetes.io/config.source: file
Status:               Running
IP:                   192.168.65.3
```

What follows next is information on the container running within the pod. The information here starts with basic information such as the container ID, images, and ports:

```
Containers:
  kube-scheduler:
    Container ID:  docker://1b7ca730cd85941a5550d816239edc14953
f07b98763751ecb1caf7dfcced087
    Image:         k8s.gcr.io/kube-scheduler:v1.15.5
    Image ID:      docker-pullable://k8s.gcr.io/kube-scheduler@
```

```
sha256:ec985e27f41e3ceec552440502dbfa723924d5e6d72fc9193d140972
e24b8b77
    Port:          <none>
    Host Port:     <none>
```

We then move on to the command that is being run within the container:

```
    Command:
      kube-scheduler
      --bind-address=127.0.0.1
      --kubeconfig=/etc/kubernetes/scheduler.conf
      --leader-elect=true
```

Now we see its current state:

```
    State:          Running
      Started:      Sun, 03 May 2020 12:11:03 +0100
    Ready:          True
    Restart Count:  0
```

Then we have some information on its utilization:

```
    Requests:
      cpu:        100m
    Liveness:     http-get http://127.0.0.1:10251/healthz
delay=15s timeout=15s period=10s #success=1 #failure=8
    Environment:  <none>
    Mounts:
      /etc/kubernetes/scheduler.conf from kubeconfig (ro)
```

Next, we are back to information on the pod. Here, we can see the current status:

```
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
```

Then, we see details on the volumes mounted by the pod and some other options such as **Quality of Service** (**QoS**):

```
Volumes:
  kubeconfig:
    Type:          HostPath (bare host directory volume)
    Path:          /etc/kubernetes/scheduler.conf
    HostPathType:  FileOrCreate
QoS Class:         Burstable
Node-Selectors:    <none>
Tolerations:       :NoExecute
```

Finally, you can see events listed:

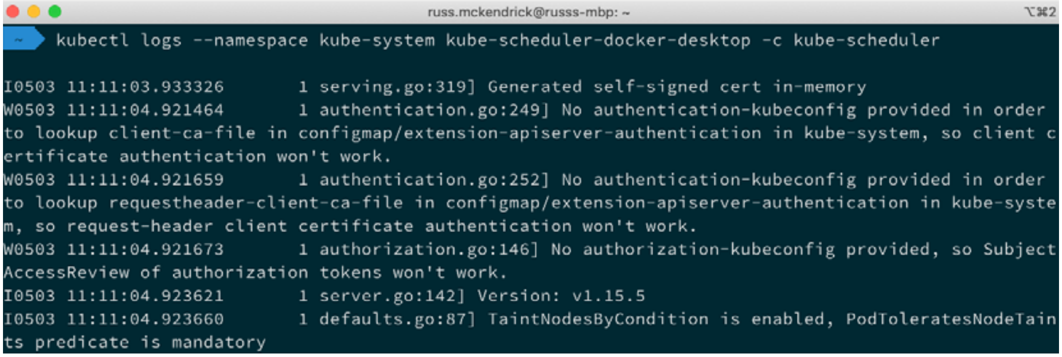
```
Events:
  Type    Reason   Age   From                      Message
  ----    ------   ----  ----                      -------
  Normal  Pulled   39m   kubelet, docker-desktop  Container
image 'k8s.gcr.io/kube-scheduler:v1.15.5' already present on
machine
  Normal  Created  39m   kubelet, docker-desktop  Created
container kube-scheduler
  Normal  Started  39m   kubelet, docker-desktop  Started
container kube-scheduler
```

As you can see, there is a lot of information about the pod, including a list of containers; we only have one called `kube-scheduler`. We can see the container ID, the image used, the flags the container was launched with, and also the data used by the Kubernetes scheduler to launch and maintain the pod.

Now that we know a container name, we can start to interact with it. For example, running the following command will print the logs for our one container:

```
$ kubectl logs kube-scheduler-docker-desktop -c kube-scheduler
--namespace kube-system
```

I got the following output:



```
~ kubectl logs --namespace kube-system kube-scheduler-docker-desktop -c kube-scheduler

I0503 11:11:03.933326       1 serving.go:319] Generated self-signed cert in-memory
W0503 11:11:04.921464       1 authentication.go:249] No authentication-kubeconfig provided in order
to lookup client-ca-file in configmap/extension-apiserver-authentication in kube-system, so client c
ertificate authentication won't work.
W0503 11:11:04.921659       1 authentication.go:252] No authentication-kubeconfig provided in order
to lookup requestheader-client-ca-file in configmap/extension-apiserver-authentication in kube-syste
m, so request-header client certificate authentication won't work.
W0503 11:11:04.921673       1 authorization.go:146] No authorization-kubeconfig provided, so Subject
AccessReview of authorization tokens won't work.
I0503 11:11:04.923621       1 server.go:142] Version: v1.15.5
I0503 11:11:04.923660       1 defaults.go:87] TaintNodesByCondition is enabled, PodToleratesNodeTain
ts predicate is mandatory
```

Figure 11.10 – Checking the logs on a container in a pod

Running the following command would fetch the logs for each container in the pod:

```
$ kubectl logs --namespace kube-system kube-scheduler-docker-
desktop
```

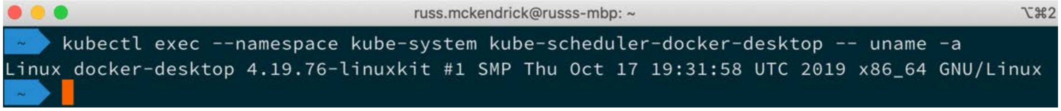
Like Docker, you can also execute commands on your pods and containers.

> **Tip**
>
> Please ensure you add the space after the -- in the following two commands. Failing to do so will result in errors.

For example, the following commands will run the `uname -a` command:

```
$ kubectl exec --namespace kube-system kube-scheduler-docker-
desktop -c kube-scheduler -- uname -a
$ kubectl exec --namespace kube-system kube-scheduler-docker-
desktop -- uname -a
```

Again, we have the option of running the command on a named container or across all containers within the pod:



```
~ kubectl exec --namespace kube-system kube-scheduler-docker-desktop -- uname -a
Linux docker-desktop 4.19.76-linuxkit #1 SMP Thu Oct 17 19:31:58 UTC 2019 x86_64 GNU/Linux
~
```

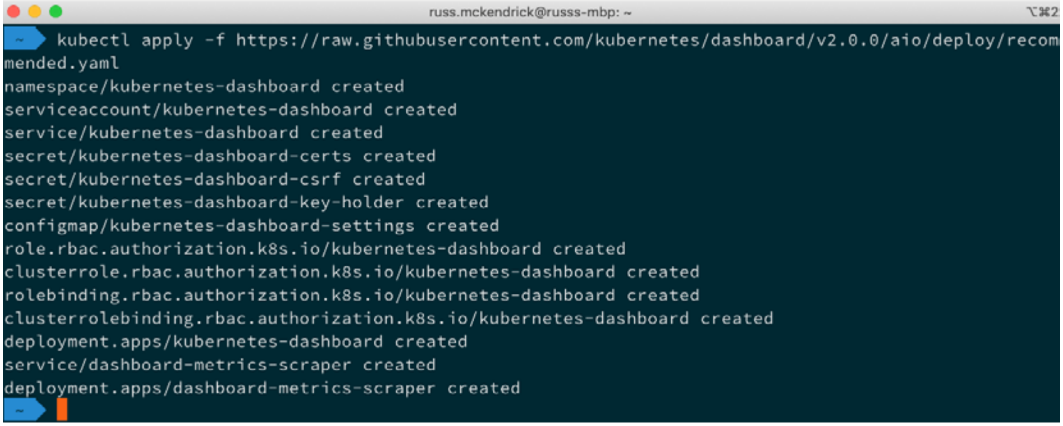
Figure 11.11 – Running a command across all the containers in a pod

Let's find out a little more about our Kubernetes cluster by installing and logging into the web-based dashboard.

While this does not ship with Docker by default, installing it using the definition file provided by the Kubernetes project is simple. We just need to run the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/
kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml
```

As soon as you run the command, you should see something like the following output:



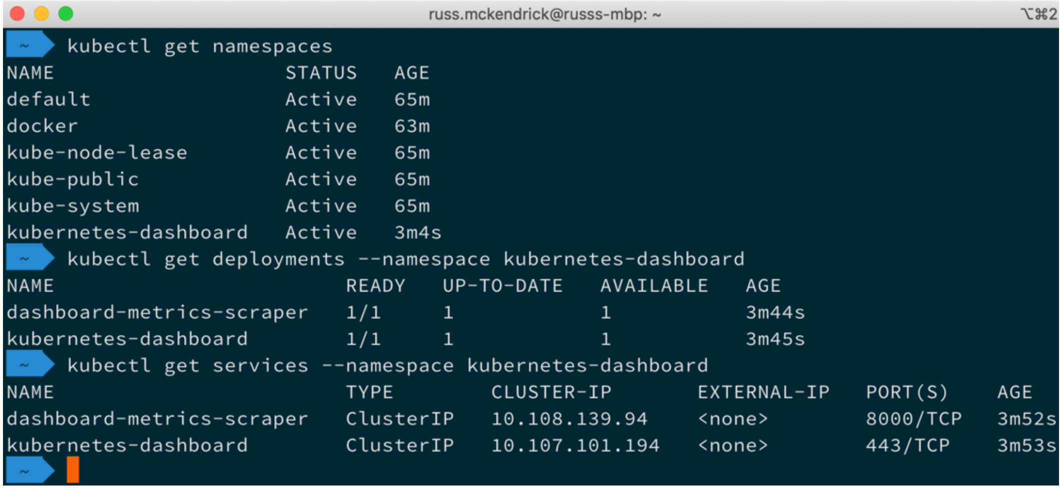
```
~ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
~
```

Figure 11.12 – Deploying the web-based dashboard

Once the services and deployments have been created, it will take a few minutes to launch. You can check on the status by running the following commands:

```
$ kubectl get namespaces
$ kubectl get deployments --namespace kubernetes-dashboard
$ kubectl get services --namespace kubernetes-dashboard
```

Once your output looks like the following, your dashboard should be installed and ready:



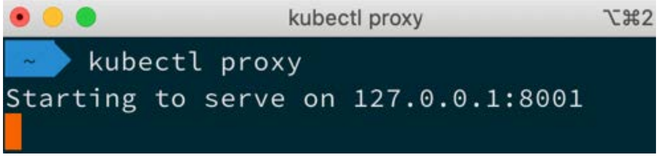
```
~ kubectl get namespaces
NAME                   STATUS   AGE
default                Active   65m
docker                 Active   63m
kube-node-lease        Active   65m
kube-public            Active   65m
kube-system            Active   65m
kubernetes-dashboard   Active   3m4s
~ kubectl get deployments --namespace kubernetes-dashboard
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
dashboard-metrics-scraper   1/1     1            1           3m44s
kubernetes-dashboard        1/1     1            1           3m45s
~ kubectl get services --namespace kubernetes-dashboard
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
dashboard-metrics-scraper   ClusterIP   10.108.139.94    <none>        8000/TCP   3m52s
kubernetes-dashboard        ClusterIP   10.107.101.194   <none>        443/TCP    3m53s
~
```

Figure 11.13 – Checking the status of the deployment

You may have noticed that the dashboard has its own namespace called `kubernetes-dashboard`. Now that we have our dashboard running, we will find a way to access it. We can do this using the inbuilt proxy service in `kubectl`. Just run the following command to start it up:

```
$ kubectl proxy
```

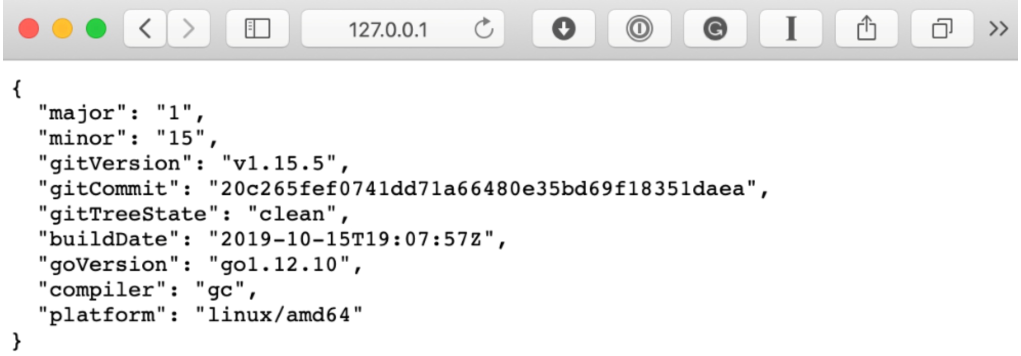
This will open a long-running foreground process:



```
~ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Figure 11.14 – Starting the proxy service

Now that the proxy service is running, opening your browser and going to `http://127.0.0.1:8001/version/` will show you some information on your cluster:



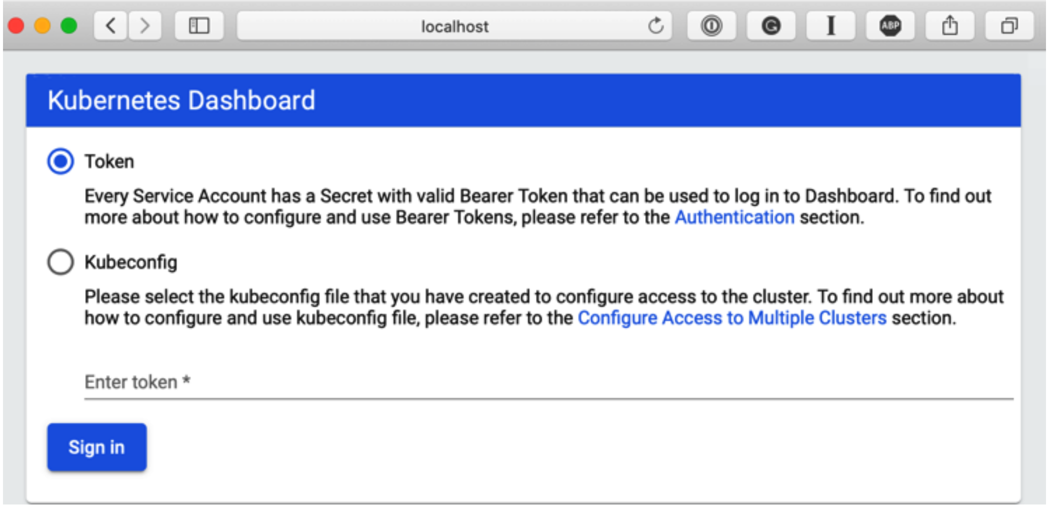
```
{
  "major": "1",
  "minor": "15",
  "gitVersion": "v1.15.5",
  "gitCommit": "20c265fef0741dd71a66480e35bd69f18351daea",
  "gitTreeState": "clean",
  "buildDate": "2019-10-15T19:07:57Z",
  "goVersion": "go1.12.10",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

Figure 11.15 – Information on the cluster

However, it's the dashboard we want to see. This can be accessed at the following URL:

```
http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/
services/https:kubernetes-dashboard:/proxy/
```

You should see something like the following screen:



Kubernetes Dashboard

● Token

Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.

○ Kubeconfig

Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.

Enter token *

Sign in

Figure 11.16 – The Dashboard login screen

As you can see, it is asking for us to log in; however, we haven't yet created any credentials, so let's do that now.

Info

A service account is a system account, which in most cases uses a token to authenticate against the Kubernetes API to perform an action. Service accounts can be used for both services running within your Kubernetes cluster, as well as in our case, where we as a user want to access to the Dashboard using an API token.

Open a new Terminal window and enter the following command to create a service account:

```
$ kubectl create serviceaccount dashboard-admin-sa
```

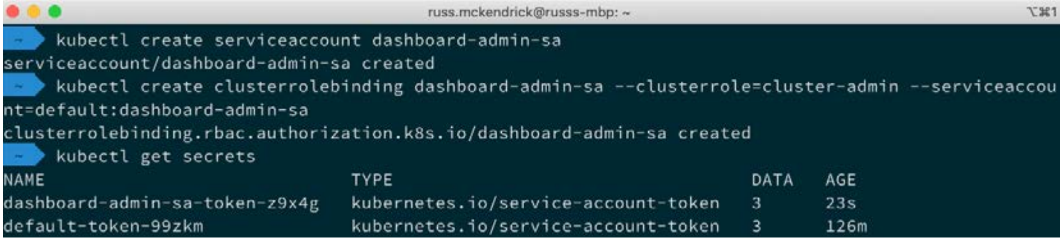
The service account will be created in the default namespace; however, that is not going to be a problem as we are now going to assign the service account the `cluster-admin` role by running the following command:

```
$ kubectl create clusterrolebinding dashboard-admin-sa
--clusterrole=cluster-admin --serviceaccount=default:dashboard-
admin-sa
```

This should have created a secret, and we can find the name of the secret by running the following command:

```
$ kubectl get secrets
```

The following Terminal output shows the steps taken so far:



```
~  kubectl create serviceaccount dashboard-admin-sa
serviceaccount/dashboard-admin-sa created
~  kubectl create clusterrolebinding dashboard-admin-sa --clusterrole=cluster-admin --serviceaccou
nt=default:dashboard-admin-sa
clusterrolebinding.rbac.authorization.k8s.io/dashboard-admin-sa created
~  kubectl get secrets
NAME                            TYPE                                  DATA   AGE
dashboard-admin-sa-token-z9x4g  kubernetes.io/service-account-token   3      23s
default-token-99zkm             kubernetes.io/service-account-token   3      126m
```

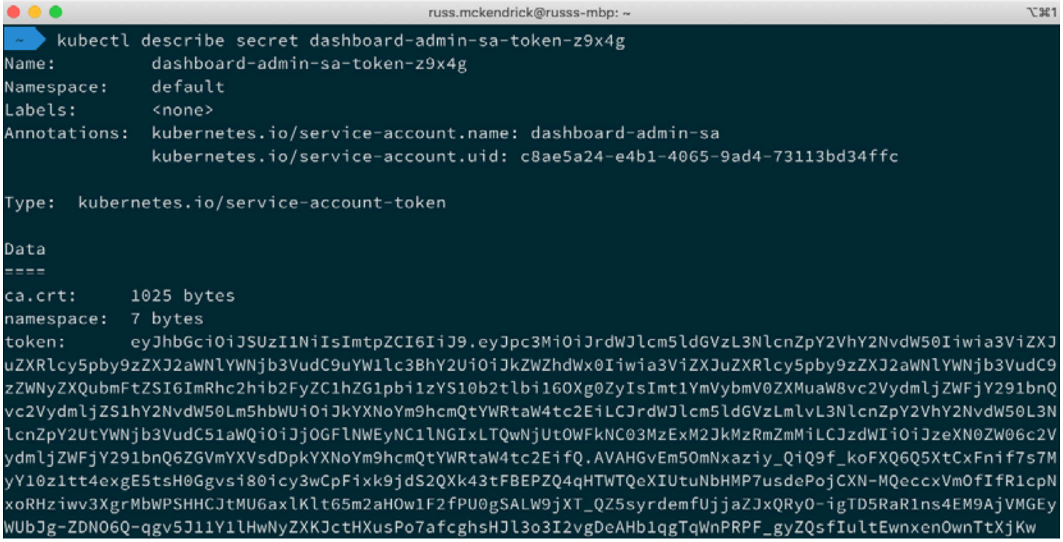
Figure 11.17 – Creating the service account, assigning permission, and viewing the secrets

Now that our service account has been created, the correct permissions have been set, and we know the name of the secret (yours will differ as the secret name is affixed with a five-character random string), we can get a copy of the token we need to log in.

We simply need to run the following command, making sure that you update the secret name to match your own:

```
$ kubectl describe secret dashboard-admin-sa-token-z9x4g
```

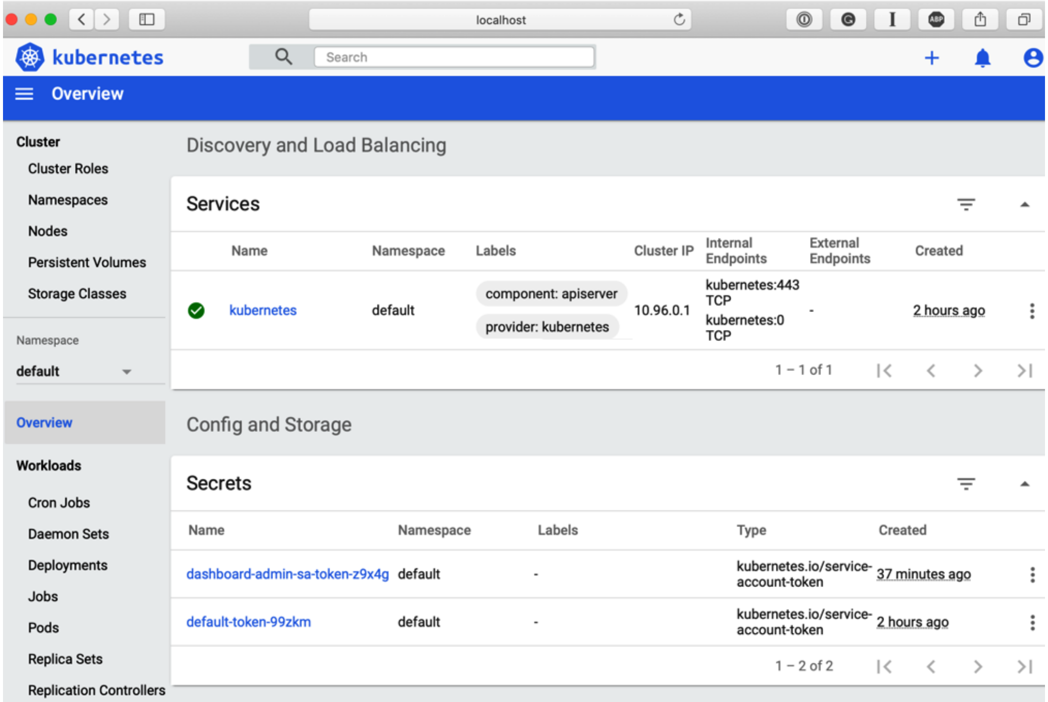
This should give you something similar to the following Terminal output:



```
~ kubectl describe secret dashboard-admin-sa-token-z9x4g
Name:         dashboard-admin-sa-token-z9x4g
Namespace:    default
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: dashboard-admin-sa
              kubernetes.io/service-account.uid: c8ae5a24-e4b1-4065-9ad4-73113bd34ffc

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1025 bytes
namespace:  7 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJ
uZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9
zZWNyZXQubmFtZSI6ImRhc2hib2FyZC1hZG1pbi1zYS10b2tlbi16OXg0ZyIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQ
vc2VydmljZS1hY2NvdW50Lm5hbWUiOiJkYXNoYm9hcmQtYWRtaW4tc2EiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3N
lcnZpY2UtYWNjb3VudC51aWQiOiJjOGFlNWEyNC1lNGIxLTQwNjUtOWFkNC03MzExM2JkMzRmZmMiLCJzdWIiOiJzeXN0ZW06c2V
ydmljZWFjY291bnQ6ZGVmYXVsdDpkYXNoYm9hcmQtYWRtaW4tc2EifQ.AVAHGvEm5OmNxaziy_QiQ9f_koFXQ6Q5XtCxFnif7s7M
yY10z1tt4exgE5tsH0Ggvsi80icy3wCpFixk9jdS2QXk43tFBEPZQ4qHTWTQeXIUtuNbHMP7usdePojCXN-MQeccxVmOfIfR1cpN
xoRHziwv3XgrMbWPSHHCJtMU6axlKlt65m2aHOw1F2fPU0gSALW9jXT_QZ5syrdemfUjjaZJxQRyO-igTD5RaR1ns4EM9AjVMGEy
WUbJg-ZDNO6Q-qgv5J11Y1lHwNyZXKJctHXusPo7afcghsHJl3o3I2vgDeAHb1qgTqWnPRPF_gyZQsfIultEwnxenOwnTtXjKw
```

Figure 11.18 – Viewing the secret

Make a note of the token and enter it on the dashboard login page in the space provided for the token and then click on the **Sign in** button. Once logged in, you will be presented with something that looks like the following page:



Services

Name	Namespace	Labels	Cluster IP	Internal Endpoints	External Endpoints	Created
kubernetes	default	component: apiserver provider: kubernetes	10.96.0.1	kubernetes:443 TCP kubernetes:0 TCP	-	2 hours ago

1 – 1 of 1

Secrets

Name	Namespace	Labels	Type	Created
dashboard-admin-sa-token-z9x4g	default	-	kubernetes.io/service-account-token	37 minutes ago
default-token-99zkm	default	-	kubernetes.io/service-account-token	2 hours ago

1 – 2 of 2

Figure 11.19 – Dashboard first login

As you can see, the dashboard uses the `default` namespace. Well, by default, clicking the namespace name will open a drop-down list containing all of the available namespaces. For now, select **All namespaces** from the top of the list, and you will notice that the view changes and a lot more information is now displayed on the overview page.

Now that we have our cluster up and running, we can now look at launching a few sample applications.

Kubernetes and other Docker tools

When we enabled Kubernetes, we selected the **Deploy Docker Stacks to Kubernetes by default** option. If you recall, in *Chapter 8*, *Docker Swarm*, we used the `docker stack` command to launch our Docker Compose files in Docker Swarm and, as you might have guessed, running those same commands will now launch our stack in our Kubernetes cluster.

The Docker Compose file we used looked like the following:

```
version: '3'
services:
  cluster:
    image: russmckendrick/cluster
    ports:
      - '80:80' deploy:
    replicas: 6
    restart_policy:
      condition: on-failure
    placement:
      constraints:
        - node.role == worker
```

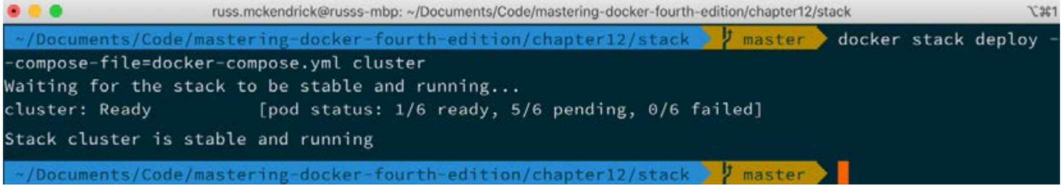
Before we launch the application on Kubernetes, we need to make a slight adjustment and remove the `placement`, which leaves our file looking like the following:

```
version: '3'
services:
  cluster:
    image: russmckendrick/cluster
    ports:
      - '80:80' deploy:
    replicas: 6
    restart_policy:
      condition: on-failure
```

Once the file has been edited, running the following command will launch the stack:

```
$ docker stack deploy --compose-file=docker-compose.yml cluster
```

As you can see, Docker waits until the stack is available before returning you to your prompt:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  docker stack deploy -
-compose-file=docker-compose.yml cluster
Waiting for the stack to be stable and running...
cluster: Ready          [pod status: 1/6 ready, 5/6 pending, 0/6 failed]
Stack cluster is stable and running
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master
```

Figure 11.20 – Launching the stack

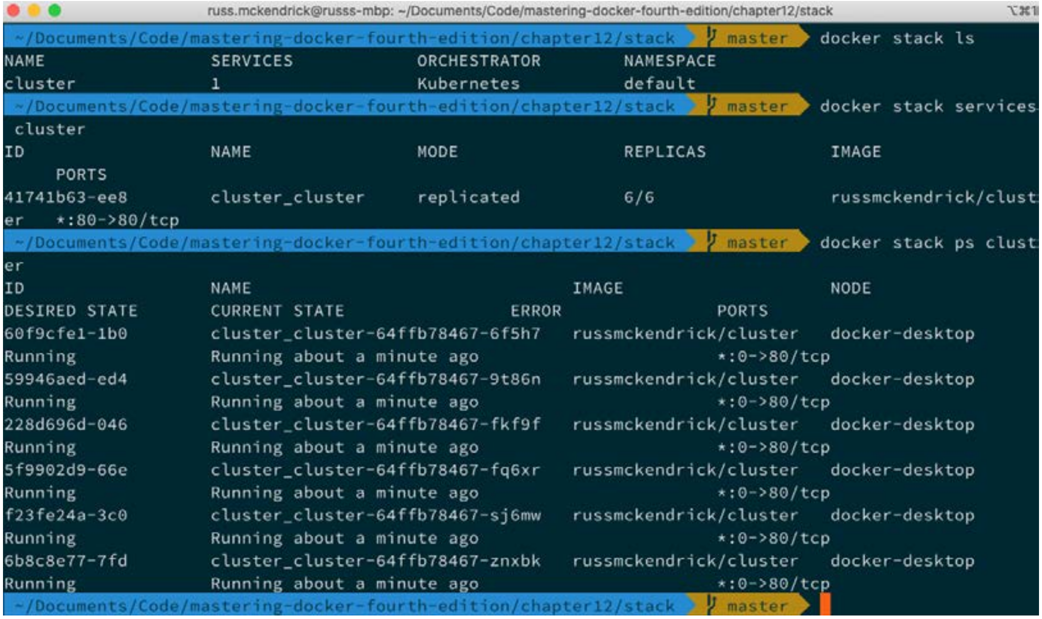
We can also run the same commands we used to view some information about our stack as we did when we launched our stack on Docker Swarm:

```
$ docker stack ls
```

```
$ docker stack services cluster
```

```
$ docker stack ps cluster
```

The Terminal output gives us similar output to when we launched the stack using a Docker Swarm cluster:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  docker stack ls
NAME      SERVICES  ORCHESTRATOR  NAMESPACE
cluster   1         Kubernetes    default
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  docker stack services
 cluster
ID        NAME             MODE        REPLICAS    IMAGE
   PORTS
41741b63-ee8   cluster_cluster   replicated   6/6   russmckendrick/clust
er   *:80->80/tcp
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  docker stack ps clust
er
ID              NAME                              IMAGE                  NODE
DESIRED STATE   CURRENT STATE                 ERROR           PORTS
60f9cfe1-1b0    cluster_cluster-64ffb78467-6f5h7  russmckendrick/cluster  docker-desktop
Running         Running about a minute ago                    *:0->80/tcp
59946aed-ed4    cluster_cluster-64ffb78467-9t86n  russmckendrick/cluster  docker-desktop
Running         Running about a minute ago                    *:0->80/tcp
228d696d-046    cluster_cluster-64ffb78467-fkf9f  russmckendrick/cluster  docker-desktop
Running         Running about a minute ago                    *:0->80/tcp
5f9902d9-66e    cluster_cluster-64ffb78467-fq6xr  russmckendrick/cluster  docker-desktop
Running         Running about a minute ago                    *:0->80/tcp
f23fe24a-3c0    cluster_cluster-64ffb78467-sj6mw  russmckendrick/cluster  docker-desktop
Running         Running about a minute ago                    *:0->80/tcp
6b8c8e77-7fd    cluster_cluster-64ffb78467-znxbk  russmckendrick/cluster  docker-desktop
Running         Running about a minute ago                    *:0->80/tcp
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master
```

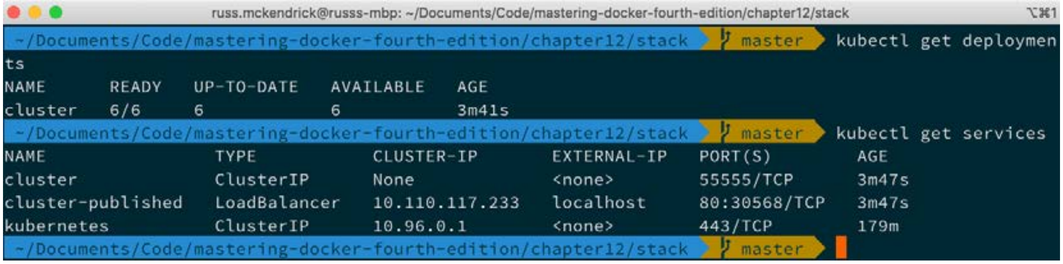
Figure 11.21 – Running the Docker stack commands

However, please note, at the time of writing there appears to be an issue with the `docker stack services` returning an error, this issue was introduced with an update to the version of Kubernetes that ships with Docker.

We can also see details using `kubectl`:

```
$ kubectl get deployments
$ kubectl get services
```

You may have noticed that this time we did not need to provide a namespace. This is because our stack was launched in the default namespace:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack        ⌥⌘1
 ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  kubectl get deploymen
ts
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
cluster    6/6     6            6           3m41s
 ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  kubectl get services
NAME                TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
cluster             ClusterIP      None             <none>        55555/TCP      3m47s
cluster-published   LoadBalancer   10.110.117.233   localhost     80:30568/TCP   3m47s
kubernetes          ClusterIP      10.96.0.1        <none>        443/TCP        179m
 ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master 
```

Figure 11.22 – Viewing details about the deployment and services

Also, when the services were listed, a **ClusterIP** and **LoadBalancer** are listed for the cluster stack. Looking at the **LoadBalancer**, you can see that the external IP is `localhost` and that the port is `80`.

Opening `http://localhost/` in our browser shows the application:

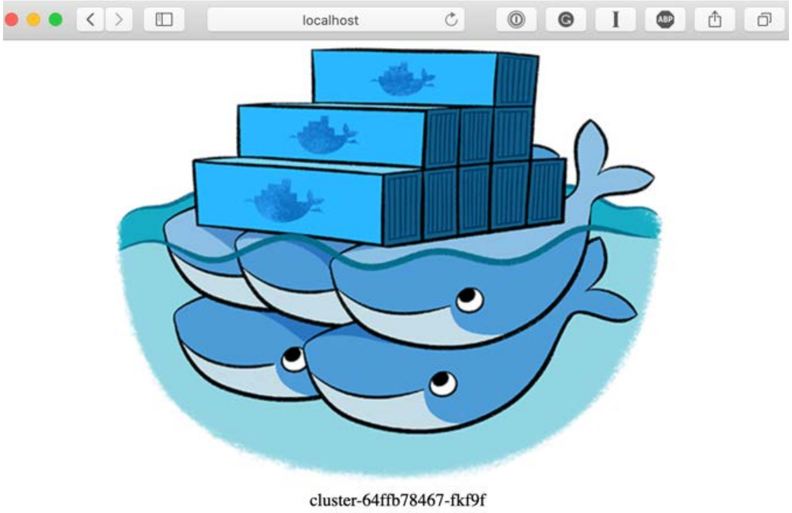
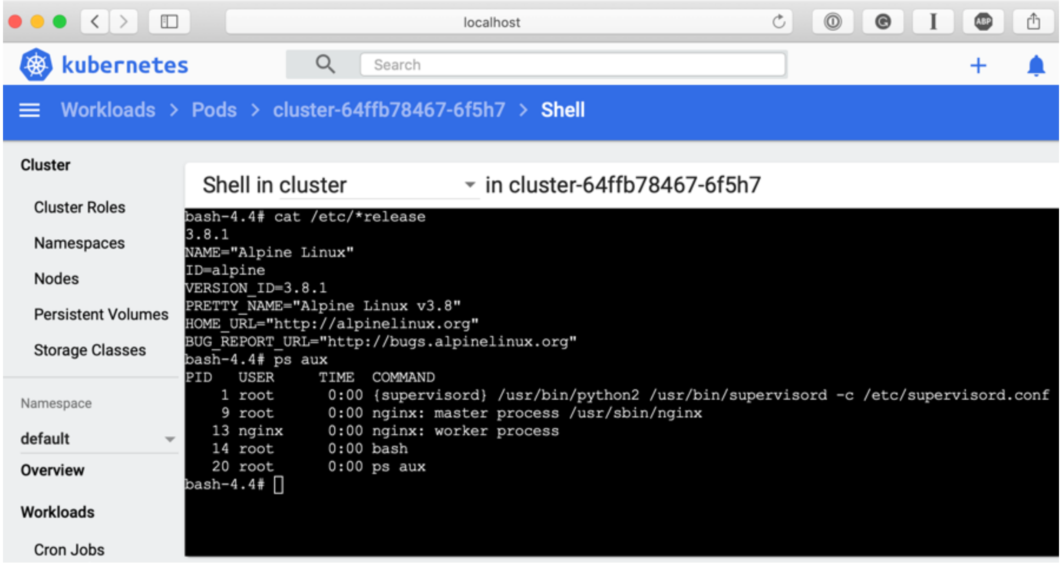


Figure 11.23 – Viewing the cluster application running in Kubernetes

If you still have the Dashboard open, you can explore your stack and even open a Terminal to one of the containers:

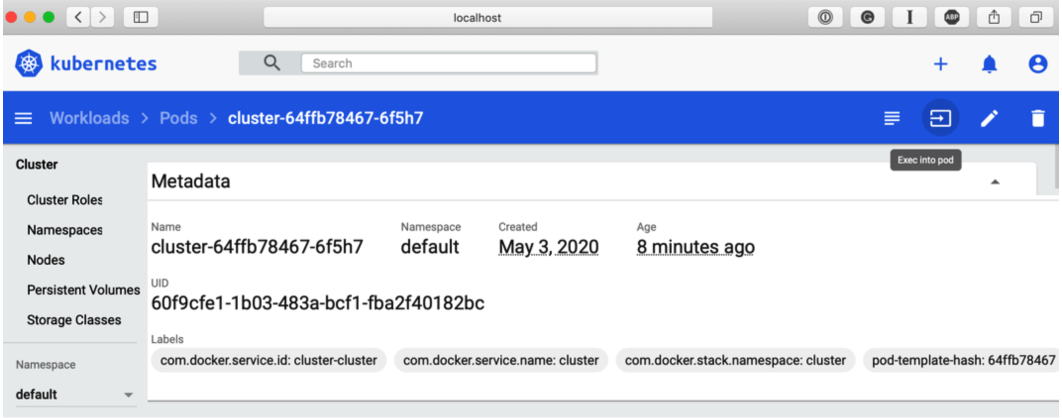


Shell in cluster in cluster-64ffb78467-6f5h7

```
bash-4.4# cat /etc/*release
3.8.1
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.8.1
PRETTY_NAME="Alpine Linux v3.8"
HOME_URL="http://alpinelinux.org"
BUG_REPORT_URL="http://bugs.alpinelinux.org"
bash-4.4# ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 {supervisord} /usr/bin/python2 /usr/bin/supervisord -c /etc/supervisord.conf
    9 root      0:00 nginx: master process /usr/sbin/nginx
   13 nginx     0:00 nginx: worker process
   14 root      0:00 bash
   20 root      0:00 ps aux
bash-4.4# 
```

Figure 11.24 – Opening a Terminal to a container

This was done by selecting one of the six pods for the cluster deployment and then clicking on the **Exec into pod** button highlighted in the following screenshot:



Metadata

Name: cluster-64ffb78467-6f5h7
Namespace: default
Created: May 3, 2020
Age: 8 minutes ago

UID: 60f9cfe1-1b03-483a-bcf1-fba2f40182bc

Labels: com.docker.service.id: cluster-cluster com.docker.service.name: cluster com.docker.stack.namespace: cluster pod-template-hash: 64ffb78467

Figure 11.25 – Exec into pod

You can remove the stack by running the following command:

```
$ docker stack rm cluster
```

One last thing…you may be thinking to yourself, 'Great, I can run my Docker Compose files anywhere on a Kubernetes cluster.' Well, that is not strictly true.

As mentioned, when we first enabled Kubernetes, there are some Docker-only components launched. These are there to make sure that Docker is integrated as tightly as possible. However, as these components won't exist in non-Docker managed clusters, you won't be able to use the `docker stack` commands.

All is not lost though. There is a tool called **Kompose** provided as part of the Kubernetes project, which can take Docker Compose files and convert them on the fly to Kubernetes definition files.

To install Kompose on macOS using Homebrew, run the following command:

```
$ brew install kompose
```

Windows 10 users can use Chocolatey.

> **Info**
>
> **Chocolatey** is a command-line-based package manager that can be used to install various software packages on your Windows machine, similar to how you can use `yum` or `apt-get` on Linux machines or `brew` on macOS.

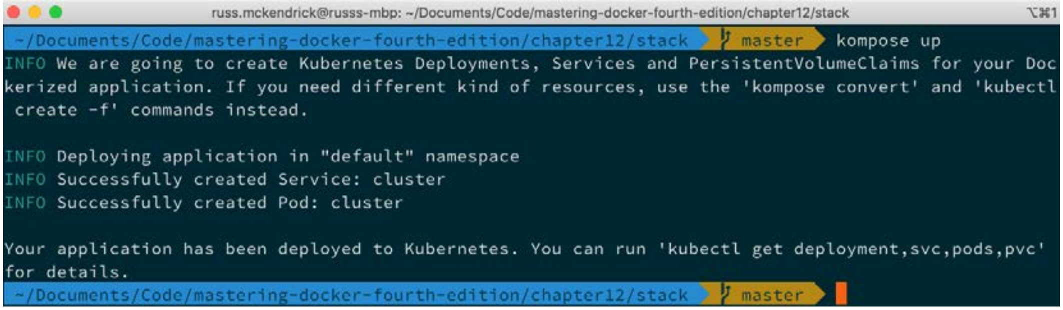
To install Kompose using Chocolatey, you can run the following command:

```
$ choco install kubernetes-kompose
```

Once it's installed, you can launch your Docker Compose file by running the following command:

```
$ kompose up
```

You will get something like the following output:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack
⌥⌘1

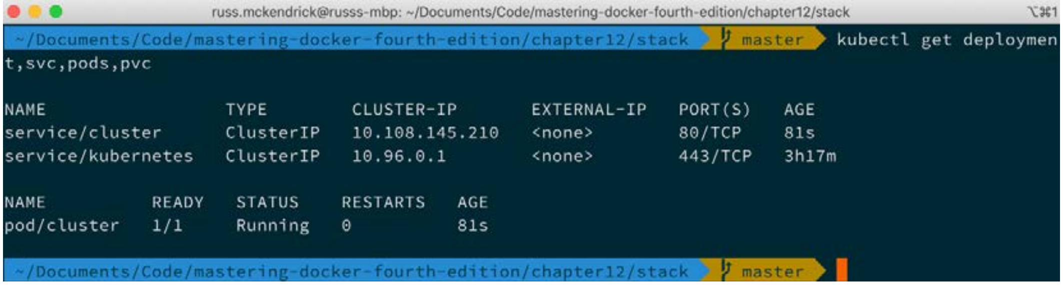
```
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  kompose up
INFO We are going to create Kubernetes Deployments, Services and PersistentVolumeClaims for your Doc
kerized application. If you need different kind of resources, use the 'kompose convert' and 'kubectl
 create -f' commands instead.

INFO Deploying application in "default" namespace
INFO Successfully created Service: cluster
INFO Successfully created Pod: cluster

Your application has been deployed to Kubernetes. You can run 'kubectl get deployment,svc,pods,pvc'
for details.
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master
```

Figure 11.26 – Running kompose up

As suggested by the output, running the following command will give you details on the service and pod we just launched:

```
$ kubectl get deployment,svc,pods,pvc
```

As you can see, our Docker Compose application is up and running:



russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack
⌥⌘1

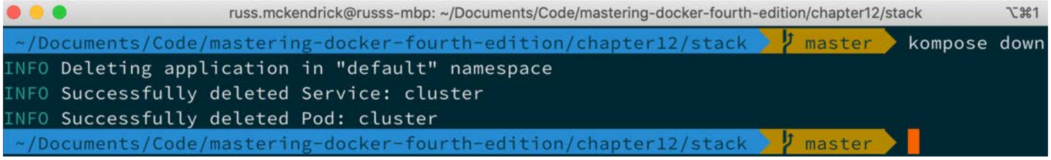
```
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  kubectl get deploymen
t,svc,pods,pvc

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
service/cluster      ClusterIP   10.108.145.210   <none>        80/TCP    81s
service/kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP   3h17m

NAME           READY   STATUS    RESTARTS   AGE
pod/cluster    1/1     Running   0          81s

~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master
```

Figure 11.27 – Checking the status of the application

You can remove the services and pods by running the following command:

```
$ kompose down
```

This should give you something like the following:



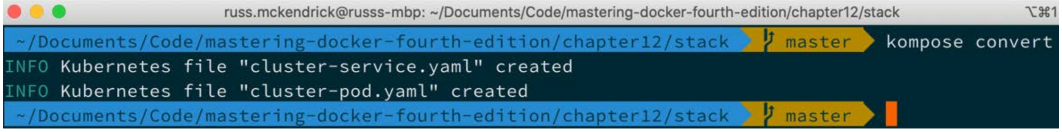
```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  kompose down
INFO Deleting application in "default" namespace
INFO Successfully deleted Service: cluster
INFO Successfully deleted Pod: cluster
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master
```

Figure 11.28 – Running kompose down

While you can use `kompose up` and `kompose down`, I would recommend generating the Kubernetes definition files and tweaking them as needed. To do this, simply run the following command:

```
$ kompose convert
```

You will notice that this command generates two files:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  kompose convert
INFO Kubernetes file "cluster-service.yaml" created
INFO Kubernetes file "cluster-pod.yaml" created
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master
```

Figure 11.29 – Running kompose convert

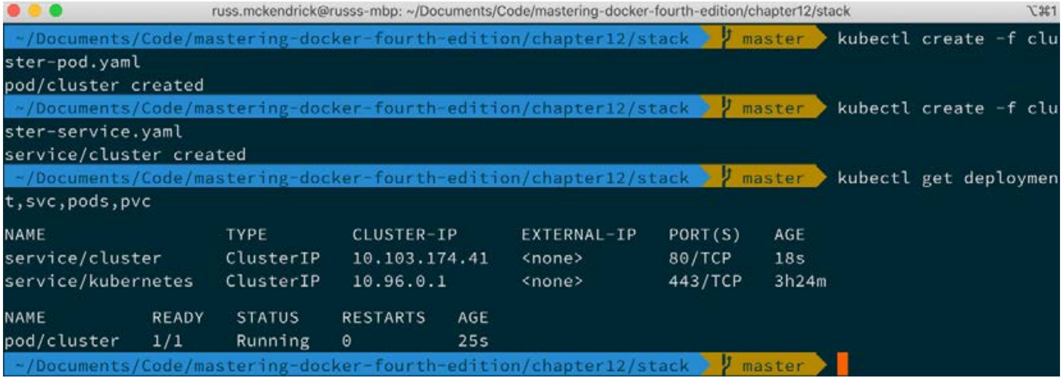
You will be able to see quite a difference between the Docker Compose file and the two files generated. The `cluster-pod.yaml` file looks like the following:

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: cluster
  name: cluster
spec:
  containers:
  - image: russmckendrick/cluster
    name: cluster
    ports:
    - containerPort: 80
    resources: {}
  restartPolicy: OnFailure
status: {}
And the cluster-service.yaml file contains the following:
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose convert
    kompose.version: 1.21.0 ()
  creationTimestamp: null
  labels:
    io.kompose.service: cluster
  name: cluster
spec:
  ports:
  - name: '80'
    port: 80
    targetPort: 80
  selector:
    io.kompose.service: cluster
status:
  loadBalancer: {}
```

You can then launch these files by running the following command:

```
$ kubectl create -f cluster-pod.yaml
$ kubectl create -f cluster-service.yaml
$ kubectl get deployment,svc,pods,pvc
```

If you are not following along, the following screenshot shows the Terminal output:



```
russ.mckendrick@russs-mbp: ~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  kubectl create -f clu
ster-pod.yaml
pod/cluster created
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  kubectl create -f clu
ster-service.yaml
service/cluster created
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master  kubectl get deploymen
t,svc,pods,pvc
NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/cluster      ClusterIP   10.103.174.41   <none>        80/TCP    18s
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   3h24m

NAME          READY   STATUS    RESTARTS   AGE
pod/cluster   1/1     Running   0          25s
~/Documents/Code/mastering-docker-fourth-edition/chapter12/stack  master
```

Figure 11.30 – Launching the application using cluster-pod.yaml and cluster-service.yaml

To remove the cluster pod and service, we just need to run the following command:

```
$ kubectl delete service/cluster pod/cluster
```

While we will continue to use this in the next two chapters, you may want to disable the Kubernetes integration within your Docker desktop installation as it does add a slight overhead to host machine when it is idle. To do this, just untick **Enable Kubernetes**. When you click **Apply**, Docker will stop all the containers it needed to run Kubernetes; it won't, however, remove the images so that when you re-enable it, it doesn't take as long.

Summary

In this chapter, we looked at Kubernetes from the point of view of Docker desktop software. There is a lot more to Kubernetes than we have covered in this chapter, so please don't think this is all there is. After discussing the origins of Kubernetes, we looked at how you can enable it on your local machine using Docker for Mac or Docker for Windows.

We then discussed some basic usage of kubectl before looking at running how we can use `docker stack` commands to launch our applications as we did for Docker Swarm.

At the end of the chapter, we discussed Kompose, which is a tool from the Kubernetes project. It helps you convert your Docker Compose files for use with Kubernetes, allowing you to get a head start on moving your applications to pure Kubernetes.

While we have referred to a Kubernetes cluster throughout this chapter, we have in actual fact been running a single node cluster, which really isn't a cluster at all.

In the next chapter, we are going to take a look at a few more options on how to launch Kubernetes locally. Here, we will welcome back Linux users and also look at options for launching more than one node.

Questions

1. True or false: When **Show system containers (advanced)** is ticked, you cannot see the images used to launch Kubernetes using the `docker image ls` command.
2. Which of the four namespaces hosts the containers used to run Kubernetes and enable support within Docker?
3. Which command would you run to find out details about a container running in a pod?
4. Which command would you use to launch a Kubernetes definition YAML file? Typically, which port does the `kubectl proxy` command open on your local machine?
5. What was the original name of Google container orchestration platform?

Further reading

Some of the Google tools, presentations, and white papers mentioned at the start of the chapter can be found here:

- cgroups: `http://man7.org/linux/man-pages/man7/cgroups.7.html`
- lmctfy: `https://github.com/google/lmctfy/`
- Containers at Scale, Joe Beda's slides from GluCon: `http://slides.eightypercent.net/GlueCon%202014%20-%20Containers%20At%20Scale.pdf`
- Large-scale cluster management at Google with Borg: `https://ai.google/research/pubs/pub43438`
- LXC: `https://linuxcontainers.org/`

You can find details on the cloud services mentioned in the chapter here:

- **Google Kubernetes Engine** (**GKE**): `https://cloud.google.com/kubernetes-engine`
- **Azure Kubernetes Service** (**AKS**): `https://azure.microsoft.com/en-gb/services/kubernetes-service/`

- Amazon **Elastic Container Service for Kubernetes** (**Amazon EKS**): `https://aws.amazon.com/eks/`
- IBM Cloud Kubernetes Service: `https://www.ibm.com/cloud/container-service`
- Oracle Container Engine for Kubernetes: `https://cloud.oracle.com/containers/kubernetes-engine`
- Kubernetes on DigitalOcean: `https://www.digitalocean.com/products/kubernetes/`

You can find Docker's announcements about Kubernetes support here:

- Docker Platform and Moby Project add Kubernetes: `https://www.docker.com/blog/top-5-blogs-2017-docker-platform-moby-project-add-kubernetes/`

Finally, the home page for Kompose can be found here:

- Kompose: `http://kompose.io/`

12
Discovering other Kubernetes options

In this chapter, we will look at alternatives to using Docker's in-built support for running your own local Kubernetes single-node and multi-node clusters.

We will be discussing and looking at the following tools:

- Minikube
- **Kind** (**Kubernetes in Docker**)
- MicroK8s
- K3s

Technical requirements

We will be using one of the tools we discussed in *Chapter 6*, *Docker Machine, Vagrant, and Multipass*, along with some standalone tools that can be used to bootstrap your own local Kubernetes installation.

Again, the screenshots in this chapter will be from my preferred operating system, macOS

Check out the following video to see the Code in Action:
`https://bit.ly/3byY1O4.`

Deploying Kubernetes using Minikube

First in our list of alternative Kubernetes cluster installers is Minikube. It was initially released in May 2016, making it the oldest of the tools we will be discussing in this chapter.

Before we look at installing Minikube, we should probably discuss why it was needed in the first place.

At the time of its original release, Kubernetes `1.2` had been out for a few months, and it was almost a year after the `1.0` release of Kubernetes.

While installing Kubernetes had become a lot simpler since the original release, it typically still boiled down to a bunch of installation scripts and step-by-step instructions that were designed to mostly bootstrap cloud-hosted clusters utilizing the cloud provider's APIs or command-line tools.

If you wanted to run an installation locally for development purposes, then you would have to either hack together your installer from existing scripts or download a Vagrant box where you would be trusting the author of the box to have built it using the best practices promoted by the **Cloud Native Computing Foundation** (**CNCF**), which was trying to ensure consistency between clusters.

Although Minikube started life as a way of creating a local Kubernetes node on just Linux and macOS hosts, Windows was soon introduced, and the project has grown to be an important part of the Kubernetes project as it is many people's first taste of interacting with Kubernetes and it forms part of the official Kubernetes documentation.

Now that we know a little about Minikube's background, let's look at getting it installed and then launching a single-node cluster.

Installing Minikube

Minikube is designed to run a single binary, and like Kubernetes itself, it is written in Go, which means that it can easily be compiled to run on the three platforms we have been looking at in this title. To start with, let's have a look at how to install Minikube on macOS.

Installing Minikube on macOS

When we have discussed installing software and tools on macOS in previous chapters, we have mentioned and used Homebrew.

Installing Minikube follows this path, meaning that it can be installed using a single command, which as you may have already guessed, looks like the following:

```
$ brew install minikube
```

Once installed, you can verify everything is working as expected by running the following:

```
$ minikube version
```

This will output the version of Minikube that was installed along with the commit ID that version was compiled from.

Installing Minikube on Windows 10

Like Homebrew on macOS, when it comes to package managers on Windows, the go-to is Chocolatey. If you have it installed, you can install Minikube using the following command:

```
$ choco install minikube
```

If you are not using Chocolatey, then you can download a Windows installer from `https://storage.googleapis.com/minikube/releases/latest/minikube-installer.exe`.

Once you have Minikube installed, you can run the following command:

```
$ minikube version
```

This will confirm the installed version and commit.

Installing Minikube on Linux

Depending on the version of Linux you are running, there are a few different ways to install Minikube.

> **Tip**
>
> Before running the commands that follow, check the project's release page on GitHub to confirm what version of Minikube to install. The page can be found at `https://github.com/kubernetes/minikube/releases/`. At the time of writing, the latest release is 1.9.2-0, so the instructions will use that version.

If you are running a Debian-based system that uses an APT such as Ubuntu or Debian itself, then you can download and install a `.deb` using the following commands:

```
$ MK_VER=1.9.2-0
$ curl -LO https://storage.googleapis.com/minikube/releases/
latest/minikube_$MK_VER_amd64.deb
$ sudo dpkg -i minikube_$MK_VER _amd64.deb
```

If you are running an RPM-based system, such as CentOS or Fedora, then you can use the following commands:

```
$ MK_VER=1.9.2-0
$ curl -LO https://storage.googleapis.com/minikube/releases/
latest/minikube-$MK_VER.x86_64.rpm
sudo rpm -ivh minikube-$MK_VER.x86_64.rpm
```

Finally, if you would prefer not to use a package manager, then you can download the latest static binary using the following commands:

```
$ curl -LO https://storage.googleapis.com/minikube/releases/
latest/minikube-linux-amd64
$ sudo install minikube-Linux-amd64 /usr/local/bin/minikube
```

Whichever way you have chosen to install Minikube, you can run the following to confirm that everything is installed and working as expected:

```
$ minikube version
```

Again, this will confirm the installed version and the commit it was compiled from.

Minikube drivers

As we have already mentioned, Minikube is a standalone static binary that helps you launch a Kubernetes node on your local machine; it does this by interacting with several hypervisors. Before we start to use Minikube, let's quickly look at the options:

- **Docker (macOS, Windows, and Linux)**: This driver uses Docker Machine to launch containers that host your Kubernetes node. On Linux, it will use just containers, and on macOS and Windows, it will launch a small virtual machine too and deploy the containers there.

- **VirtualBox (macOS, Windows, and Linux)**: This driver works across three operating systems and will launch a virtual machine and then configure your node.
- **HyperKit (macOS)**: This uses the hypervisor built into macOS to host a virtual machine where your node will be configured.
- **Hyper-V (Windows 10 Pro)**: This uses the native hypervisor that is built into Windows 10 Professional to host a virtual machine where your node will be launched and configured.
- **KVM2 (Linux)**: This driver uses **(Kernel-based Virtual Machine KVM)** to launch a virtual machine.
- **Podman (Linux)**: This experimental driver uses a Docker replacement called Podman to launch containers that make up your cluster.
- **Parallels (macOS)**: Uses the macOS-only Parallels to host a virtual machine where your node will be launched and configured.
- **VMware (macOS)**: Uses the macOS-only VMware Fusion to host a virtual machine where your node will be launched and configured.
- **None (Linux)**: This final option does away with containers and virtual machines launched in your local hypervisor; it simply installs your cluster node directly on the host you are running.

If you have made it this far through the book, then you will have already installed the perquisites for at least one of the drivers mentioned above on your host machine. Minikube is intelligent enough to figure out the best driver to use on your host so we should be able to proceed without worrying that things won't work.

However, should you get an error because a support driver for your chosen host operating system is not found, then see the *Further reading* section for links to each of the preceding projects.

> **Tip**
>
> Should you find yourself without a supported driver install, I would recommend installing VirtualBox as it has the least requirements and complexity out of all of the drivers.

Now that we have Minikube installed, and we have looked at the drivers that it needs to launch and configure your cluster node, we can get to work and launch our cluster node.

Launching a cluster node using Minikube

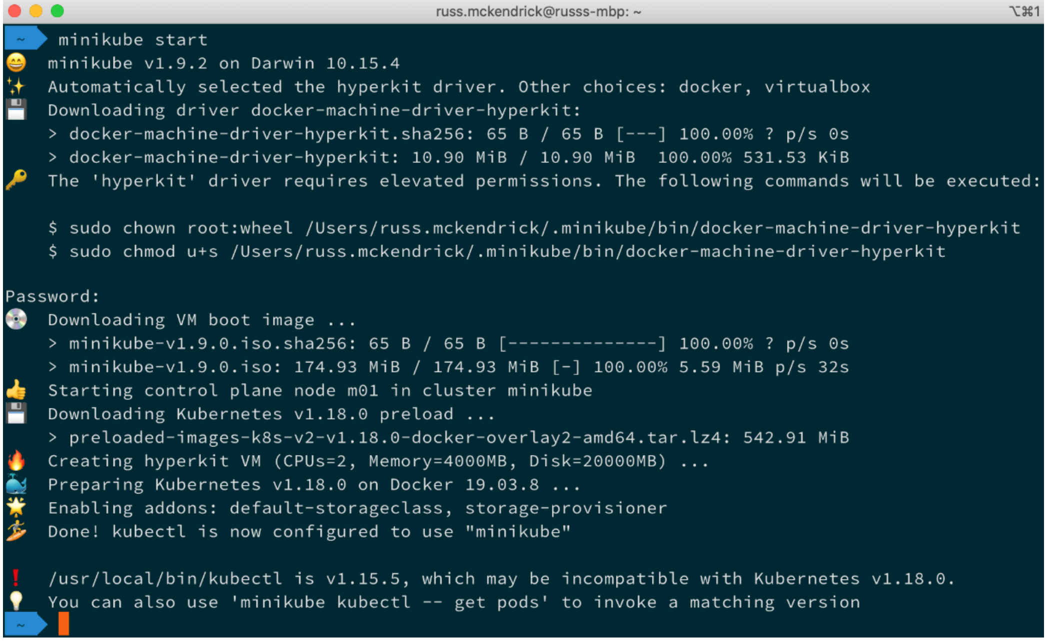
It doesn't matter what your host operating system, launching a Kubernetes cluster node using Minikube is a single, simple command:

```
$ minikube start
```

When you first run the command, it may take several minutes after detecting your local configuration as it has to download and configure various supporting tools, virtual machine images, and containers (depending on the driver it is using).

You may also find that it asks for passwords for parts of the installation that require elevated privileges. These steps are a one-off and once you have launched your initial Kubernetes cluster node, subsequent cluster node launches should be a lot quicker.

The following Terminal output shows my initial Minikube cluster node launching:



```
~  minikube start
😄  minikube v1.9.2 on Darwin 10.15.4
✨  Automatically selected the hyperkit driver. Other choices: docker, virtualbox
💾  Downloading driver docker-machine-driver-hyperkit:
    > docker-machine-driver-hyperkit.sha256: 65 B / 65 B [---] 100.00% ? p/s 0s
    > docker-machine-driver-hyperkit: 10.90 MiB / 10.90 MiB  100.00% 531.53 KiB
🔑  The 'hyperkit' driver requires elevated permissions. The following commands will be executed:

    $ sudo chown root:wheel /Users/russ.mckendrick/.minikube/bin/docker-machine-driver-hyperkit
    $ sudo chmod u+s /Users/russ.mckendrick/.minikube/bin/docker-machine-driver-hyperkit

Password:
💿  Downloading VM boot image ...
    > minikube-v1.9.0.iso.sha256: 65 B / 65 B [--------------] 100.00% ? p/s 0s
    > minikube-v1.9.0.iso: 174.93 MiB / 174.93 MiB [-] 100.00% 5.59 MiB p/s 32s
👍  Starting control plane node m01 in cluster minikube
💾  Downloading Kubernetes v1.18.0 preload ...
    > preloaded-images-k8s-v2-v1.18.0-docker-overlay2-amd64.tar.lz4: 542.91 MiB
🔥  Creating hyperkit VM (CPUs=2, Memory=4000MB, Disk=20000MB) ...
🐳  Preparing Kubernetes v1.18.0 on Docker 19.03.8 ...
🌟  Enabling addons: default-storageclass, storage-provisioner
🏄  Done! kubectl is now configured to use "minikube"

❗  /usr/local/bin/kubectl is v1.15.5, which may be incompatible with Kubernetes v1.18.0.
💡  You can also use 'minikube kubectl -- get pods' to invoke a matching version
~  
```

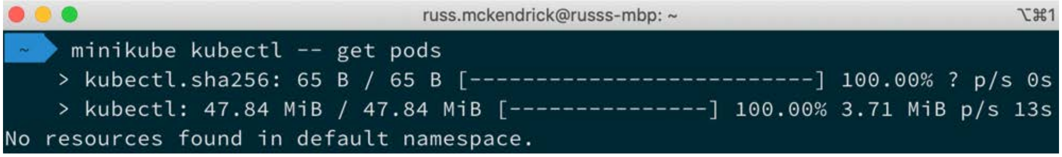
Figure 12.1 – Launching a Kubernetes cluster node on macOS

Once your cluster node is up and running, you may notice a message at the bottom of the output highlighting that the version of `kubectl` is out of date and may not be compatible with the version of Kubernetes that is installed on the Minikube launched cluster node.

Luckily, you can work around this by prefixing our `kubectl` commands with `minikube`, and adding – after. This will download and install a compatible version of `kubectl`, but it will be isolated in our Minikube workspace:

```
$ minikube kubectl -- get pods
```

You should see something like the following output:



```
~  minikube kubectl -- get pods
    > kubectl.sha256: 65 B / 65 B [-------------------------] 100.00% ? p/s 0s
    > kubectl: 47.84 MiB / 47.84 MiB [---------------] 100.00% 3.71 MiB p/s 13s
No resources found in default namespace.
```

Figure 12.2 – Initializing a compatible version of kubectl using Minikube

So, why did we get this message in the first place? Well, the Docker for Mac installation comes bundled with a version of `kubectl` that supports the version of Kubernetes natively supported within Docker, which at the time of writing is version `1.15.5`, whereas Minikube has pulled down the latest stable release of Kubernetes, which is `1.18.0`.

One of the advantages of using Minikube over the native Docker-supported Kubernetes is that you can run different versions of Kubernetes and easily isolate the versions of the supporting tools required to interact with various versions by prefixing your `kubectl` commands with `minikube` and adding `--` immediately after.

This is great when you have to test how your applications will react when it is time to update your production Kubernetes clusters.

Interacting with your Kubernetes cluster node

Now that we have our cluster node up and running, we can run through some common commands and launch a test application.

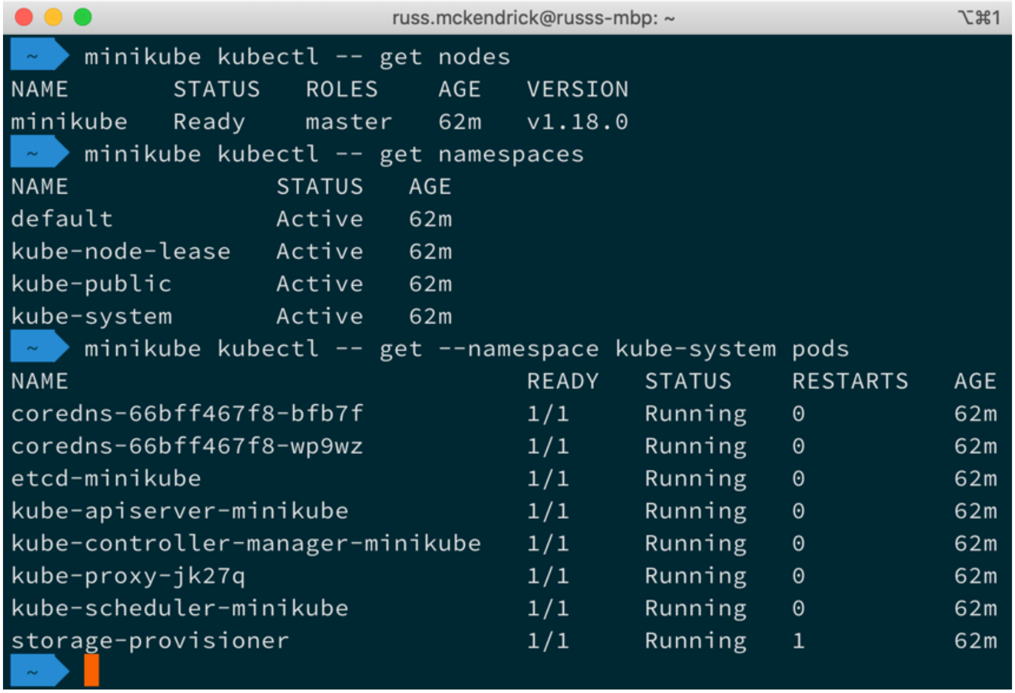
To start with, let's get a little information about our cluster node. In *Chapter 11, Docker and Kubernetes*, we ran the following commands to get information on the cluster node and namespaces:

```
$ kubectl get nodes
$ kubectl get namespaces
```

Let's run them again, remembering to prefix `minikube` before and `--` after each command so that they now look like this:

```
$ minikube kubectl -- get nodes
$ minikube kubectl -- get namespaces
$ minikube kubectl -- get --namespace kube-system pods
```

Take a look at the following Terminal output:



```
~  minikube kubectl -- get nodes
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   62m   v1.18.0
~  minikube kubectl -- get namespaces
NAME              STATUS   AGE
default           Active   62m
kube-node-lease   Active   62m
kube-public       Active   62m
kube-system       Active   62m
~  minikube kubectl -- get --namespace kube-system pods
NAME                               READY   STATUS    RESTARTS   AGE
coredns-66bff467f8-bfb7f           1/1     Running   0          62m
coredns-66bff467f8-wp9wz           1/1     Running   0          62m
etcd-minikube                      1/1     Running   0          62m
kube-apiserver-minikube            1/1     Running   0          62m
kube-controller-manager-minikube   1/1     Running   0          62m
kube-proxy-jk27q                   1/1     Running   0          62m
kube-scheduler-minikube            1/1     Running   0          62m
storage-provisioner                1/1     Running   1          62m
~
```

Figure 12.3 – Getting information on the cluster node

The output we get when running the commands is similar to that we got when we executed the equivalent commands in *Chapter 11*, *Docker and Kubernetes*, although the pods listed for the `kube-system` namespace are different.

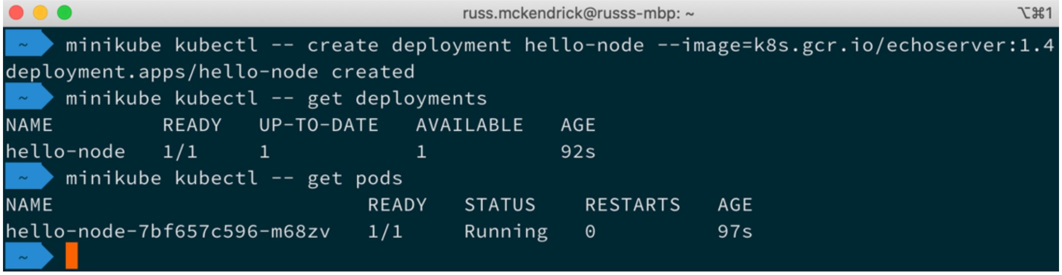
Next up, we can launch a test application by running the following command:

```
$ minikube kubectl -- create deployment hello-node --image=k8s.
gcr.io/echoserver:1.4
```

Once the deployment has been created, you can view its status and the pod involved by running the following:

```
$ minikube kubectl -- get deployments
$ minikube kubectl -- get pods
```

You should see something similar to the following Terminal output:



```
~ minikube kubectl -- create deployment hello-node --image=k8s.gcr.io/echoserver:1.4
deployment.apps/hello-node created
~ minikube kubectl -- get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
hello-node   1/1     1            1           92s
~ minikube kubectl -- get pods
NAME                          READY   STATUS    RESTARTS   AGE
hello-node-7bf657c596-m68zv   1/1     Running   0          97s
~
```

Figure 12.4 – Launching an application deployment and checking the status

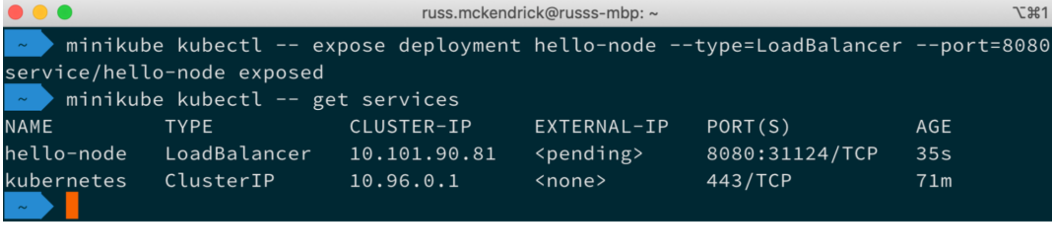
Now that our application deployment has been launched, we need a way of interacting with it. To do this, we can run the following command, which will launch a load balancer service on port `8080`:

```
$ minikube kubectl -- expose deployment hello-node
--type=LoadBalancer --port=8080
```

Once the service has been exposed, we can run the following command to get more information on running services:

```
$ minikube kubectl -- get services
```

If our cluster node was hosted on a public cloud, the command will let us know what the external IP address of the service is, however, when we run the command, we get the following output:



```
~ minikube kubectl -- expose deployment hello-node --type=LoadBalancer --port=8080
service/hello-node exposed
~ minikube kubectl -- get services
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
hello-node   LoadBalancer   10.101.90.81   <pending>     8080:31124/TCP   35s
kubernetes   ClusterIP      10.96.0.1      <none>        443/TCP          71m
~
```

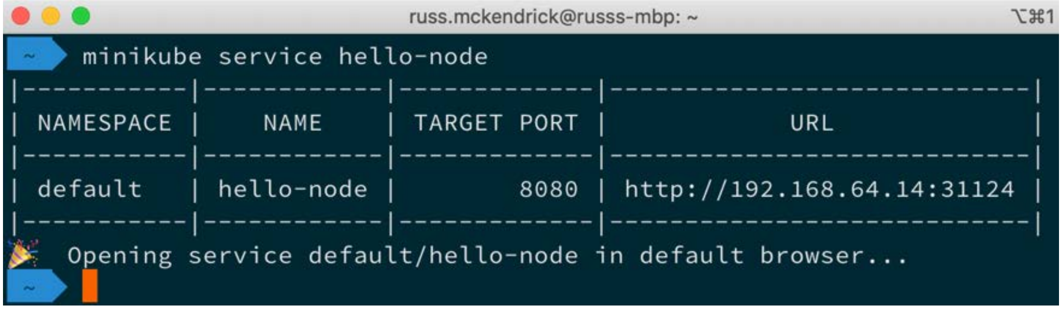
Figure 12.5 – Exposing the service on port 8080

As you can see, `EXTERNAL-IP` is listed as `<pending>`, so how do we access the application we deployed?

To this, we need to use the following command:

```
$ minikube service hello-node
```

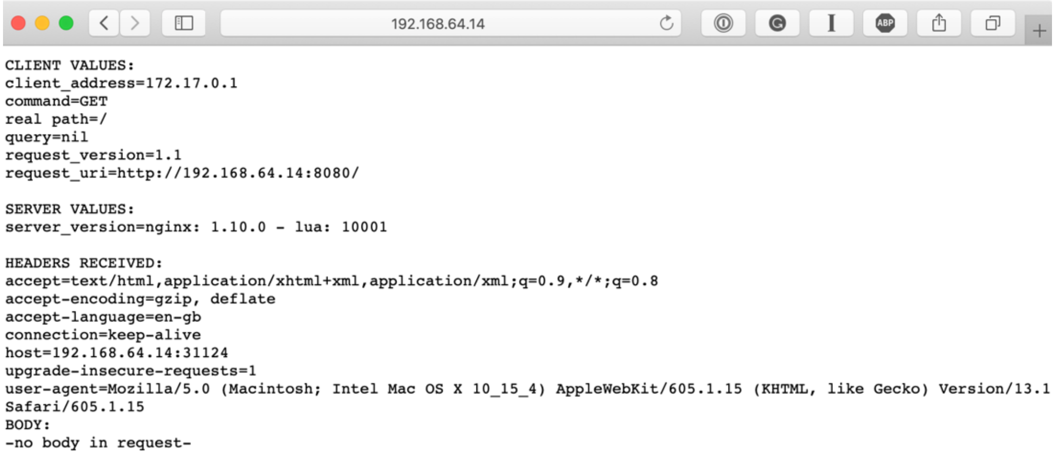
This command will open the `hello-node` service in the default browser on your machine as well as printing the URL you can access the service on in the Terminal:



```
~ minikube service hello-node
|-----------|------------|-------------|-----------------------------|
| NAMESPACE |    NAME    | TARGET PORT |             URL             |
|-----------|------------|-------------|-----------------------------|
| default   | hello-node |        8080 | http://192.168.64.14:31124 |
|-----------|------------|-------------|-----------------------------|
🎉  Opening service default/hello-node in default browser...
~
```

Figure 12.6 – Opening the hello-node service

The `hello-node` application simply echoes back the headers sent by your browser:



```
CLIENT VALUES:
client_address=172.17.0.1
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://192.168.64.14:8080/

SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept-encoding=gzip, deflate
accept-language=en-gb
connection=keep-alive
host=192.168.64.14:31124
upgrade-insecure-requests=1
user-agent=Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/13.1
Safari/605.1.15
BODY:
-no body in request-
```

Figure 12.7 – Viewing the hello-node application in a browser

Before we finish looking at Minikube, we should look at launching the cluster application we have been running in the previous chapters. To do this, run the following commands to create the deployment, expose the service, and get some information on the running pods and services:

```
$ minikube kubectl -- create deployment cluster
--image=russmckendrick/cluster:latest
$ minikube kubectl -- expose deployment cluster
--type=LoadBalancer --port=80
$ minikube kubectl -- get svc,pods
```

Once launched, we can list the URLs of all of the exposed services on our cluster node by running the following:

```
$ minikube service list
```

Opening the URL for the cluster service in a browser should show the application as expected:

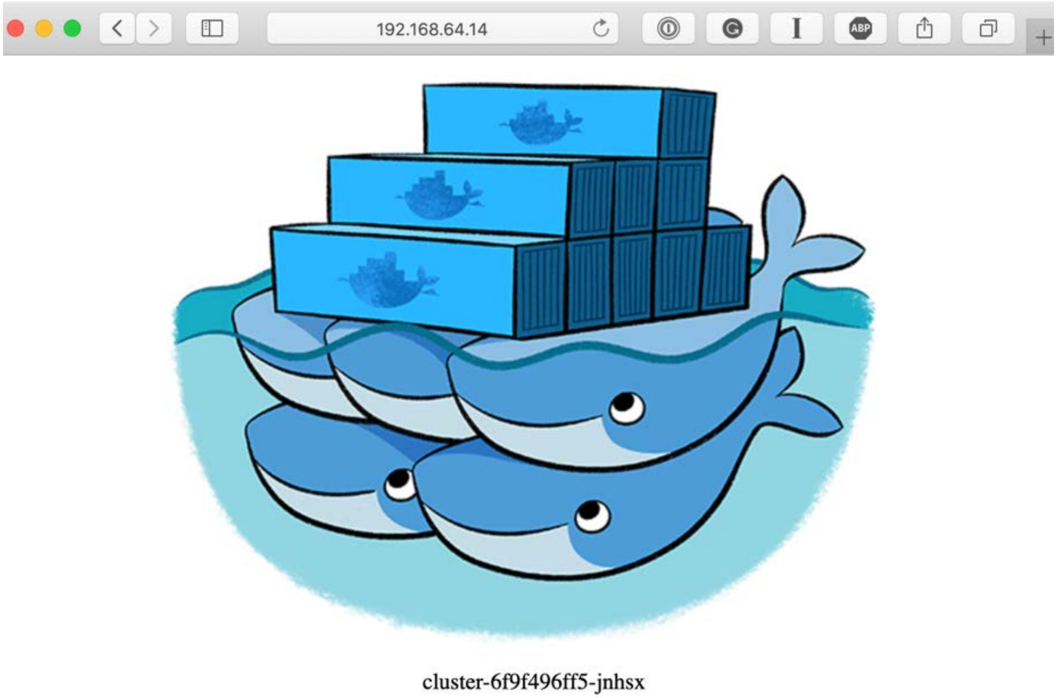


Figure 12.8 – Viewing the cluster application in a browser

Before we move on to the next tool, let's have a quick look at some of the other Minikube commands.

Managing Minikube

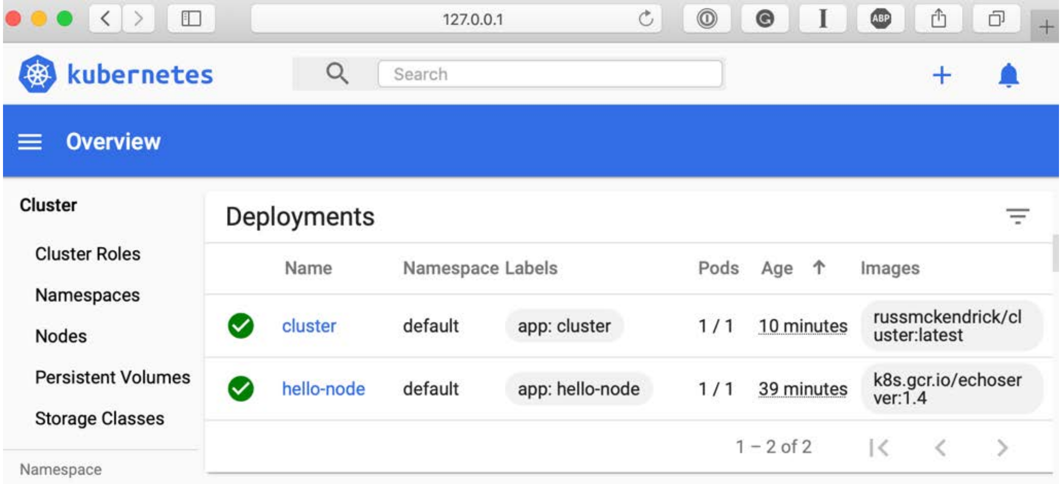
There are a few other commands we can use to manage our cluster node. The first of which is a command that allows you to quickly access the Kubernetes dashboard.

Minikube dashboard

Accessing the Kubernetes dashboard is a little more straight forward using Minikube; in fact, it is just a single command:

```
$ minikube dashboard
```

This will enable the dashboard, launch the proxy, and then open the dashboard in your default browser:



Deployments

	Name	Namespace	Labels	Pods	Age ↑	Images
✓	cluster	default	app: cluster	1 / 1	10 minutes	russmckendrick/cluster:latest
✓	hello-node	default	app: hello-node	1 / 1	39 minutes	k8s.gcr.io/echoserver:1.4

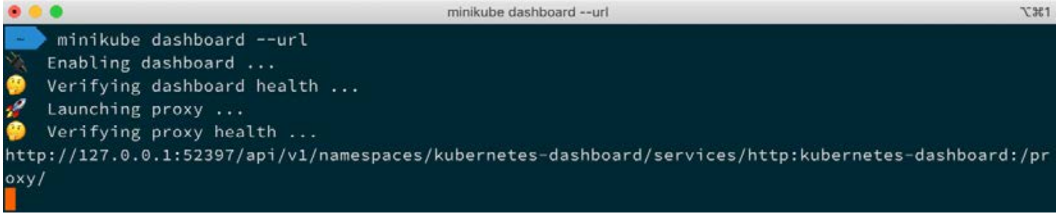
1 – 2 of 2

Figure 12.9 – Viewing the Kubernetes dashboard

There is no additional work to configure authentication needed this time. Also, you can use the following command to just get the URL you need to access the dashboard:

```
$ minikube dashboard
```

This will return something like the following output:



```
~ minikube dashboard --url
  Enabling dashboard ...
  Verifying dashboard health ...
  Launching proxy ...
  Verifying proxy health ...
http://127.0.0.1:52397/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/pr
oxy/
```

Figure 12.10 – Getting the dashboard URL

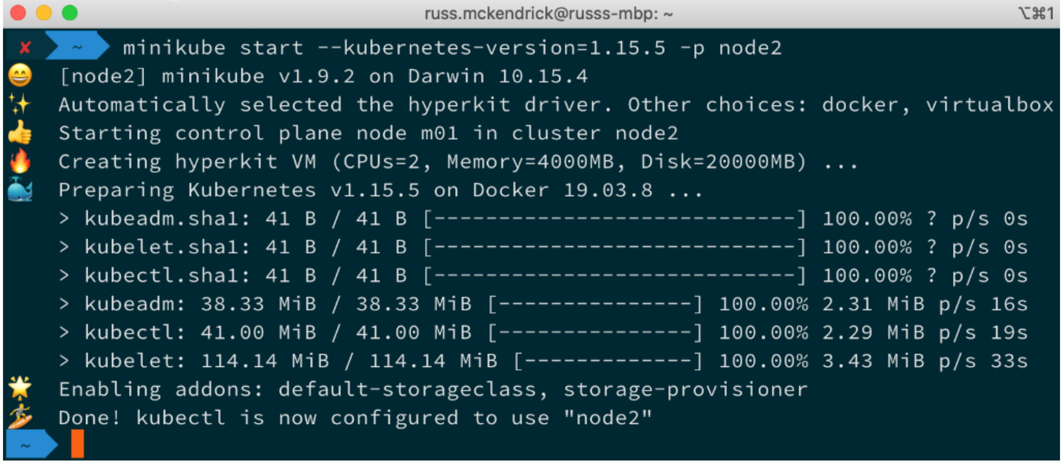
Now let's look at launching a different version of Kubernetes.

Minikube start with Kubernetes version

When we first launched our cluster node, I mentioned it is possible to launch a cluster node with a different version of Kubernetes. Using the following command, we can launch a second cluster node running Kubernetes `v1.15.5`, which is the same version currently supported by Docker for Mac:

```
$ minikube start --kubernetes-version=1.15.5 -p node2
```

This command should show you something similar to the output we originally saw when we first launched our cluster node:



```
✘ ~ minikube start --kubernetes-version=1.15.5 -p node2
  [node2] minikube v1.9.2 on Darwin 10.15.4
  Automatically selected the hyperkit driver. Other choices: docker, virtualbox
  Starting control plane node m01 in cluster node2
  Creating hyperkit VM (CPUs=2, Memory=4000MB, Disk=20000MB) ...
  Preparing Kubernetes v1.15.5 on Docker 19.03.8 ...
    > kubeadm.sha1: 41 B / 41 B [----------------------------] 100.00% ? p/s 0s
    > kubelet.sha1: 41 B / 41 B [----------------------------] 100.00% ? p/s 0s
    > kubectl.sha1: 41 B / 41 B [----------------------------] 100.00% ? p/s 0s
    > kubeadm: 38.33 MiB / 38.33 MiB [---------------] 100.00% 2.31 MiB p/s 16s
    > kubectl: 41.00 MiB / 41.00 MiB [---------------] 100.00% 2.29 MiB p/s 19s
    > kubelet: 114.14 MiB / 114.14 MiB [-------------] 100.00% 3.43 MiB p/s 33s
  Enabling addons: default-storageclass, storage-provisioner
  Done! kubectl is now configured to use "node2"
~
```

Figure 12.11 – Installing Kubernetes v1.15.5 on a second cluster node

As you can see, the process was pretty painless, and this time we did not get a complaint about potential compatibility issues with the locally installed `kubectl`.

We can view the different cluster nodes by running the following:

```
$ minikube profile list
```

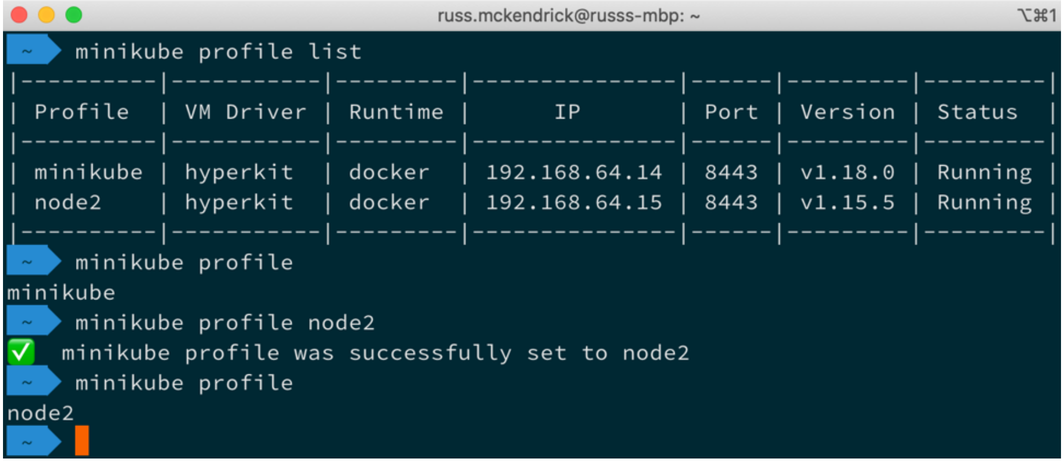
This will list the available cluster nodes. To check which one you are currently using, run the following:

```
$ minikube profile
```

To switch to a different cluster node run this:

```
$ minikube profile node2
```

Make sure you use the cluster node name you want to switch to:



```
~ minikube profile list
|----------|-----------|---------|---------------|------|---------|---------|
| Profile  | VM Driver | Runtime |      IP       | Port | Version | Status  |
|----------|-----------|---------|---------------|------|---------|---------|
| minikube | hyperkit  | docker  | 192.168.64.14 | 8443 | v1.18.0 | Running |
| node2    | hyperkit  | docker  | 192.168.64.15 | 8443 | v1.15.5 | Running |
|----------|-----------|---------|---------------|------|---------|---------|
~ minikube profile
minikube
~ minikube profile node2
✅  minikube profile was successfully set to node2
~ minikube profile
node2
~
```

Figure 12.12 – Switching cluster nodes

Now we can switch cluster nodes, what about accessing them?

Minikube SSH

Although you shouldn't need to access the cluster node itself, you can run this to get shell access to the currently selected cluster node:

```
$ minikube ssh
```

This is useful if you are curious to see what is going on under the hood.

Minikube stop and delete

The final commands we are going to look at are the ones to stop our cluster nodes or remove them altogether; you may have already guessed the command to stop a node:

```
$ minikube stop
```

This will stop the currently selected cluster node, and it can easily be started up with the following:

```
$ minikube start
```

You can add `-p <profile name>` to stop or start another cluster node. To remove the cluster node, you can run this:

```
$ minikube delete
```

This will remove the currently selected cluster node. Again, you can add `-p <profile name>` to interact with any other cluster node. Running the following will delete all of the cluster nodes:

```
$ minikube delete --all
```

There is no warning or "are you sure?" prompt when running the `minikube delete` command, so please be careful.

Minikube summary

As I am sure you will agree, Minikube has a wealth of options and is extremely straightforward to use. With it being part of the Kubernetes project itself, you will find that it is always a more up-to-date Kubernetes experience than you would get enabling Kubernetes in Docker for Mac or Docker for Windows, and it also has Linux support.

Finally, you also get an environment that is a lot closer to a CNCF-compliant Kubernetes cluster running on a public cloud, which we'll be looking at in more detail in *Chapter 13, Running Kubernetes in Public Clouds*.

Minikube takes a similar approach to Docker in that it deploys a small managed virtual machine to run your environment. The next tool we are going to look at takes a more modern, and at the time of writing, experimental approach to running Kubernetes.

Deploying Kubernetes using kind

The next tool we are going to look at is Kind, which is short for **Kubernetes in Docker**. This is exactly what you think it would be, based on the name – a Kubernetes cluster node condensed down into a container. Kind is a very recent project – so recent, in fact, that at the time of writing it is still undergoing a lot of active development. Because of this, we aren't going to spend too much time on it.

Installing Kind

Like Minikube, Kind is distributed as a single static binary – meaning its installation is very similar.

To install it on macOS, we need to run the following:

```
$ brew install kind
```

On Windows, run this:

```
$ choco install kind
```

Finally, on Linux, you can run the following:

```
$ KIND_VER=v0.8.1
$ curl -LO https://kind.sigs.k8s.io/dl/$KIND_VER/kind-$(uname)-
amd64
$sudo install kind-$(uname)-amd64 /usr/local/bin/kind
```

The release page to confirm the version number can be found at `https://github.com/kubernetes-sigs/kind/releases/`.

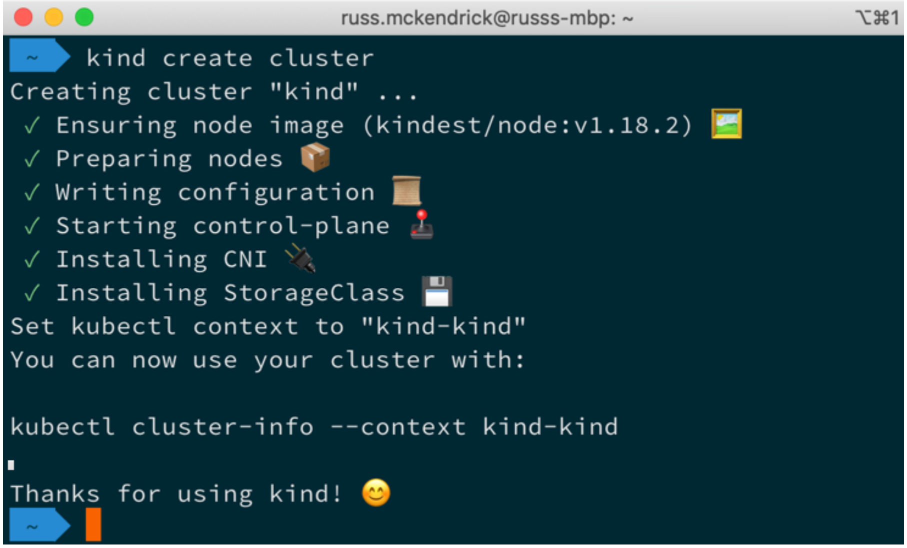
As we already have Docker installed, we don't need to worry about drivers, hypervisors, or anything to run a supporting virtual machine as Kind will simply use our local Docker installation.

Launching a Kind cluster

Once the kind binary has been installed, launching a cluster node is a very simple process; just run the following command:

```
$ kind create cluster
```

As you can see from the following Terminal output, this will download the necessary images and take care of configuring both the cluster and creating a context on your local Docker host so that you can use `kubectl` to interact with the cluster node:



```
~ kind create cluster
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.18.2) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Thanks for using kind! 😊
~
```

Figure 12.13 – Launching a cluster node with Kind

Now that we have our cluster node up and running, let's do something with it.

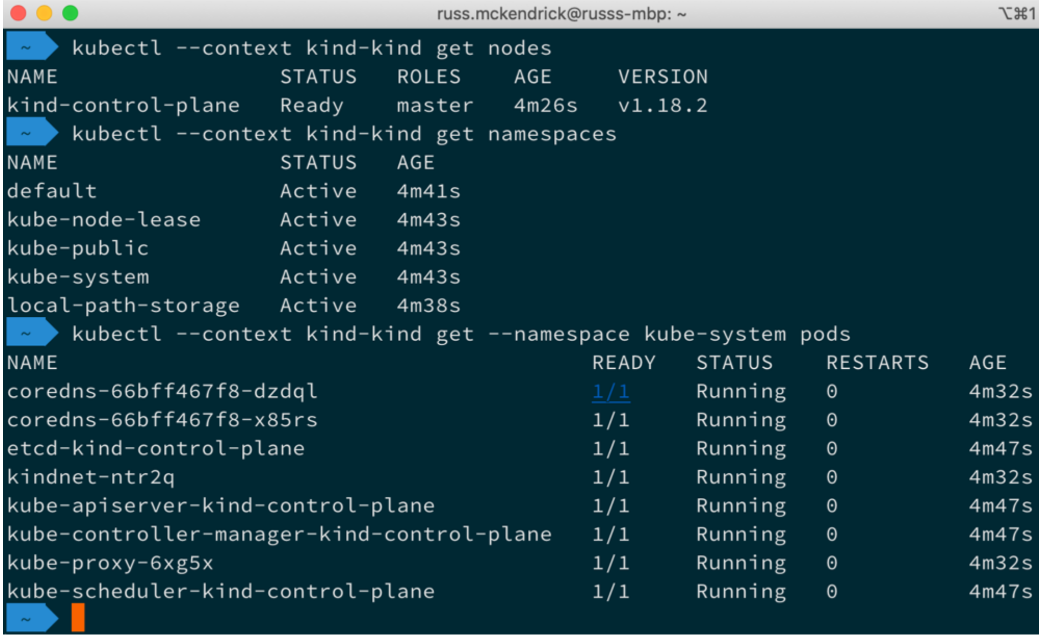
Interacting with your Kubernetes cluster node

Now that we have our cluster node up and running, we can rerun the commands and launch a test application as we did in the previous section of the chapter.

This time, we will be using the `kubectl` command on our Docker host, rather than using a wrapper; however, we will be adding a context to make sure that our Kind Kubernetes node cluster is used. This means that the commands we need to run look like the following:

```
$ kubectl --context kind-kind get nodes
$ kubectl --context kind-kind get namespaces
$ kubectl --context kind-kind get --namespace kube-system pods
```

As you can see from the following Terminal output, we again see similar output to the last time we queried our nodes, namespaces, and system pods:



```
~ kubectl --context kind-kind get nodes
NAME                 STATUS   ROLES    AGE     VERSION
kind-control-plane   Ready    master   4m26s   v1.18.2
~ kubectl --context kind-kind get namespaces
NAME                 STATUS   AGE
default              Active   4m41s
kube-node-lease      Active   4m43s
kube-public          Active   4m43s
kube-system          Active   4m43s
local-path-storage   Active   4m38s
~ kubectl --context kind-kind get --namespace kube-system pods
NAME                                         READY   STATUS    RESTARTS   AGE
coredns-66bff467f8-dzdql                     1/1     Running   0          4m32s
coredns-66bff467f8-x85rs                     1/1     Running   0          4m32s
etcd-kind-control-plane                      1/1     Running   0          4m47s
kindnet-ntr2q                                1/1     Running   0          4m32s
kube-apiserver-kind-control-plane            1/1     Running   0          4m47s
kube-controller-manager-kind-control-plane   1/1     Running   0          4m47s
kube-proxy-6xg5x                             1/1     Running   0          4m32s
kube-scheduler-kind-control-plane            1/1     Running   0          4m47s
~
```

Figure 12.14 – Viewing the nodes, namespaces, and system pods

Next, let's deploy the **hello-node** application again, using the following commands:

```
$ kubectl --context kind-kind create deployment hello-node
--image=k8s.gcr.io/echoserver:1.4
$ kubectl --context kind-kind get deployments
$ kubectl --context kind-kind get pods
$ kubectl --context kind-kind expose deployment hello-node
--type=LoadBalancer --port=8080
$ kubectl --context kind-kind get services
```

So far so good, you may be thinking, but unfortunately that is about as far as we can take the installation with the current configuration – while we can deploy pods and services, Kind does not come with an Ingress controller by default at the moment.

To enable an Ingress controller, we first have to delete our cluster. To do that, run the following command:

```
$ kind delete cluster
```

Once the cluster has been deleted, we can re-launch it with the Ingress configuration enabled. The configuration is too long to list here, but you can find a copy of it in the repository that accompanies this book in the `chapter12/kind` folder. To launch the cluster with the config, change to the `chapter12/kind` folder in your Terminal and run the following command:

```
$ kind create cluster --config cluster-config.yml
```

Once launched, the next step is to enable the NGINX Ingress controller. You will need to execute the following command to do this:

```
$ kubectl --context kind-kind apply -f https://raw.
githubusercontent.com/kubernetes/ingress-nginx/master/deploy/
static/provider/kind/deploy.yaml
```

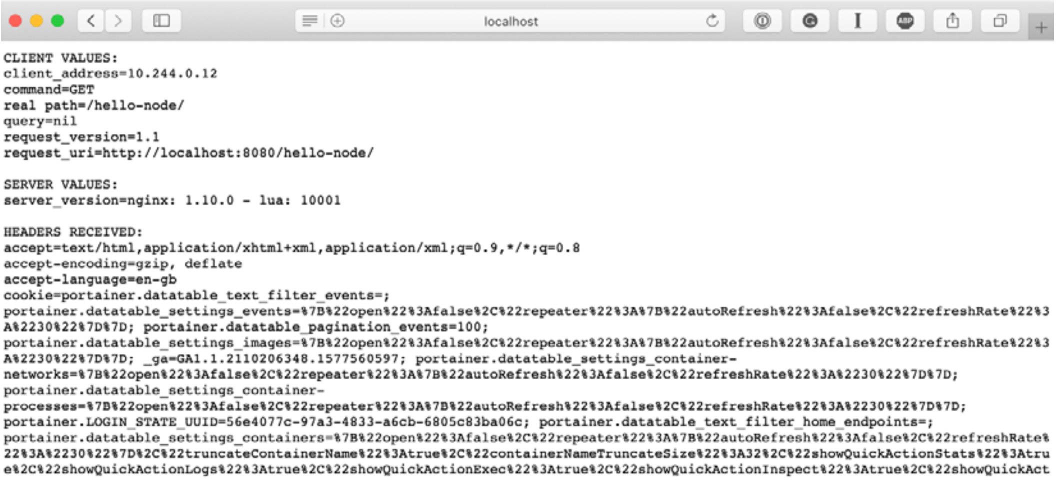
That will configure the cluster to use the NGINX Ingress controller. The controller itself take a minute or two to launch – you can check the status by running the following:

```
$ kubectl --context kind-kind get --namespace ingress-nginx
pods
```

Once you have the `Ingress-nginx-controller` pod ready and running, you can then relaunch the `hello-node` application using the `hello-node.yml` file in the `chapter12/kind` folder with this:

```
$ kubectl --context kind-kind apply -f hello-node.yml
```

Once launched, you should be able to access the `hello-node` application at `http://localhost/hello-node/` as seen in the following screenshot:



```
CLIENT VALUES:
client_address=10.244.0.12
command=GET
real path=/hello-node/
query=nil
request_version=1.1
request_uri=http://localhost:8080/hello-node/

SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept-encoding=gzip, deflate
accept-language=en-gb
cookie=portainer.datatable_text_filter_events=;
portainer.datatable_settings_events=%7B%22open%22%3Afalse%2C%22repeater%22%3A%7B%22autoRefresh%22%3Afalse%2C%22refreshRate%22%3
A%2230%22%7D%7D; portainer.datatable_pagination_events=100;
portainer.datatable_settings_images=%7B%22open%22%3Afalse%2C%22repeater%22%3A%7B%22autoRefresh%22%3Afalse%2C%22refreshRate%22%3
A%2230%22%7D%7D; _ga=GA1.1.2110206348.1577560597; portainer.datatable_settings_container-
networks=%7B%22open%22%3Afalse%2C%22repeater%22%3A%7B%22autoRefresh%22%3Afalse%2C%22refreshRate%22%3A%2230%22%7D%7D;
portainer.datatable_settings_container-
processes=%7B%22open%22%3Afalse%2C%22repeater%22%3A%7B%22autoRefresh%22%3Afalse%2C%22refreshRate%22%3A%2230%22%7D%7D;
portainer.LOGIN_STATE_UUID=56e4077c-97a3-4833-a6cb-6805c83ba06c; portainer.datatable_text_filter_home_endpoints=;
portainer.datatable_settings_containers=%7B%22open%22%3Afalse%2C%22repeater%22%3A%7B%22autoRefresh%22%3Afalse%2C%22refreshRate%
22%3A%2230%22%7D%2C%22truncateContainerName%22%3Atrue%2C%22containerNameTruncateSize%22%3A32%2C%22showQuickActionStats%22%3Atru
e%2C%22showQuickActionLogs%22%3Atrue%2C%22showQuickActionExec%22%3Atrue%2C%22showQuickActionInspect%22%3Atrue%2C%22showQuickAct
```

Figure 12.15 – The output of the hello-node application

Once finished, you can delete the cluster with the `kind delete cluster` command.

Kind summary

So, you may be thinking to yourself, what is the point of Kind – why on earth would you want to run a Kubernetes cluster in a single container? Well, its main use is to test Kubernetes itself; however, it could be used to test deployments as part of a continuous delivery or continuous deployment pipeline where you need to test your Kubernetes definition files are working as expected and your application launches without any problems.

As it stands at the moment, Kind is probably too slow and too heavy in development to be used to develop on.

Let's move onto the next tool, which takes us back to running a virtual machine to deploy our local Kubernetes cluster on.

Deploying Kubernetes using MicroK8s

Next up, we have MicroK8s by Canonical who, as you may remember from *Chapter 6, Docker Machine, Vagrant, and Multipass*, are the creators of Multipass and also the Linux distribution Ubuntu.

The mantra of the MicroK8s project is to provide a lightweight Kubernetes node with only a minimal number of basic services enabled by default while providing additional services as needed via plugins.

Installing MicroK8s

Unlike Minikube and Kind, the standalone binary for MicroK8s only works on Linux-based machines. Because of this, we are going to use Multipass to launch a virtual machine and use that as our installation target.

To launch the virtual machine, we need to run the following:

```
$ multipass launch --name microk8s
```

Once the virtual machine is up and running, we can then enable and install MicroK8s with the following command:

```
$ multipass exec microk8s -- \
    /bin/bash -c 'sudo snap install microk8s --classic'
```

Once installed, it will take a little while for MicroK8s to start up and the cluster node to be ready to use. Run the following command to poll the status and check MicroK8s is up and running:

```
$ multipass exec microk8s -- \
    /bin/bash -c 'sudo microk8s status --wait-ready'
```

Next up, as MicroK8s by default is a minimal Kubernetes cluster node, we need to enable the `dns` and `Ingress` plugins, which will allow us to access our applications when we launch them:

```
$ multipass exec microk8s -- \
    /bin/bash -c 'sudo microk8s enable dns ingress'
```

Once enabled, the final thing we need to do is get a copy of the configuration on our host machine. To do this, run the following command:

```
$ multipass exec microk8s -- \
    /bin/bash -c 'sudo microk8s.config' > microk8s.yml
```

This will leave us with a file called `microk8s.yml` in the current directory on our host machine. We can now use this configuration when we run `kubectl` to access our newly launched and configured MicroK8s Kubernetes cluster node.

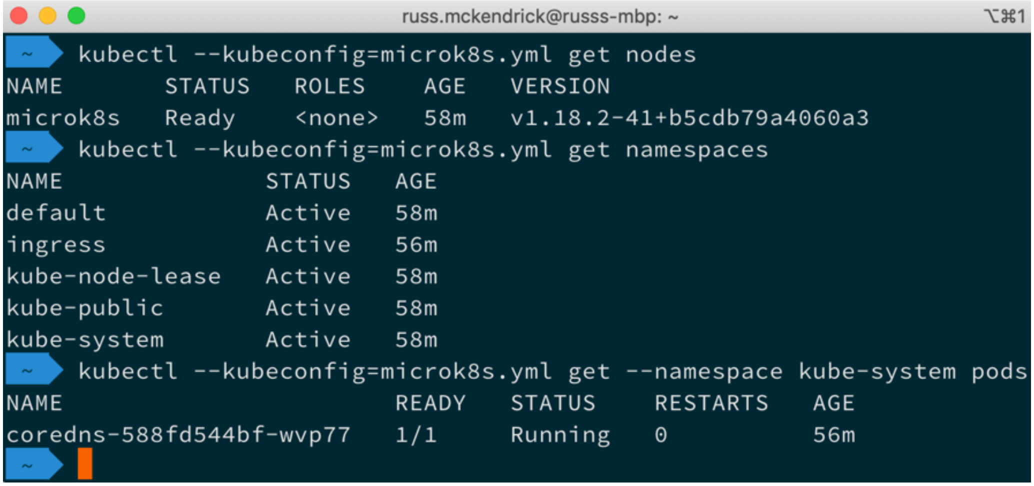
Interacting with your Kubernetes cluster node

Now that we have our cluster node up and running, we can rerun the commands and launch a test application as we did in the previous section of the chapter.

As already mentioned, this time we will be using the `kubectl` command on our host machine and pass the flag to make sure it uses the `microk8s.yml` config file. This means that the commands we need to run look like the following:

```
$ kubectl --kubeconfig=microk8s.yml get nodes
$ kubectl --kubeconfig=microk8s.yml get namespaces
$ kubectl --kubeconfig=microk8s.yml get --namespace kube-system
pods
```

As you can see from the following Terminal output, we again see similar output to the last time we queried our nodes and namespaces until we get to the system pods:



```
~ kubectl --kubeconfig=microk8s.yml get nodes
NAME       STATUS   ROLES    AGE   VERSION
microk8s   Ready    <none>   58m   v1.18.2-41+b5cdb79a4060a3
~ kubectl --kubeconfig=microk8s.yml get namespaces
NAME              STATUS   AGE
default           Active   58m
ingress           Active   56m
kube-node-lease   Active   58m
kube-public       Active   58m
kube-system       Active   58m
~ kubectl --kubeconfig=microk8s.yml get --namespace kube-system pods
NAME                      READY   STATUS    RESTARTS   AGE
coredns-588fd544bf-wvp77  1/1     Running   0          56m
~
```

Figure 12.16 – Viewing the nodes, namespaces, and system pods

The reason why we can't see any of the Pods outside of the `coredns` one is that the user attached to the configuration we downloaded doesn't have the necessary permissions to do so. Although, that's not a problem for us as we don't need to touch those pods.

Next up, we can launch the `hello-node` application. This time, we will use the YAML file straight out of the GitHub repository by running the following command:

```
$ kubectl --kubeconfig=microk8s.yml apply -f https://raw.
githubusercontent.com/PacktPublishing/Mastering-Docker-Fourth-
Edition/master/chapter12/kind/hello-node.yml
```

The reason we are using this file is that it has the definition for the Ingress controller, which means we just need to get the IP address of our MicroK8s cluster node and then enter that URL into our browser. To get the IP address, run the following:

```
$ multipass info microk8s
```

Once you know the IP address, which should be listed as IPv4, open a browser and go to `https://<IP Address>/hello-node/`. In my case, the URL was `https://192.168.64.16/hello-node/`. You will note that, this time, we are using HTTPS rather than HTTP. That is because the Ingress controller we enabled installs a self-signed certificate and redirects all traffic to HTTPS. Depending on the browser settings, you may be asked to accept or install the certificate when going to the page.

Once finished, you can delete the cluster node using the `multipass delete --purge microk8s` command. Also, don't forget to remove the `microk8s.yml` file. Once you have deleted the cluster, the config file would not work if you were to relaunch it.

MicroK8s summary

MicroK8s delivers on its promise of a small, lightweight, but still functional and extendable Kubernetes cluster controller. Coupled with Multipass, you are easily able to spin up a virtual machine and quickly bootstrap your Kubernetes cluster node on your local workstation.

Also, Canonical has made sure that MicroK8s isn't just for local use; the cluster node itself can be considered to be production-ready, meaning that it is perfect for running Kubernetes on both edge and IoT devices – both of which traditionally have lower specifications and would normally run a single node.

The final tool we are going to look at will allow us to run multiple Kubernetes nodes locally, taking us closer to what a production environment might look like.

Deploying Kubernetes using K3s

The final tool we are going to take a look at is K3s from Rancher. Like MicroK8s, K3s is a lightweight Kubernetes distribution designed for edge and IoT devices. This again makes it perfect for local development too as K3s is also a certified Kubernetes distribution – as is Docker, Kind, and MicroK8s.

> **Important note**
>
> You may be wondering why on earth it is called K3s. There is some logic behind it. As Rancher's main design aim for K3s was to produce something with half of the memory footprint of a typical Kubernetes distribution, they decided that as Kubernetes is a 10-letter word but is stylized as K8s, then their distribution would be half the size – 5 letters – and would, therefore, be stylized as K3s. However, there is no long-form for K3s and nor is there an official pronunciation.

Finally, K3s supports multi-node clusters, so we are going to look at building a three-node cluster. The commands we'll be using in this section of the chapter will cover macOS and Linux systems as we will be creating environment variables and using non-Windows tools to streamline the installation process as much as possible.

Installing K3s

Like MicroK8s, we are going to be using Multipass to launch our host machines. To do this, run the following commands:

```
$ multipass launch --name k3smaster
$ multipass launch --name k3snode1
$ multipass launch --name k3snode2
```

Once we have our three VMs up and running, we can configure the master node by running the following:

```
$ multipass exec k3smaster -- \
   /bin/bash -c 'curl -sfL https://get.k3s.io | K3S_KUBECONFIG_
MODE='644' sh -'
```

Once we have the master node up and running, we need a little information to be able to bootstrap the two remaining nodes.

The first piece of information we need is the URL of the master node. To create an environment variable, we can run the following command:

```
$ K3SMASTER='https://$(multipass info k3smaster | grep 'IPv4' |
 awk -F' ' '{print $2}'):6443'
```

Now we have the URL of the master node, we need to grab the access token. To do that, run the following:

```
$ K3STOKEN='$(multipass exec k3smaster -- /bin/bash -c 'sudo
cat /var/lib/rancher/k3s/server/node-token')'
```

Now we have the two bits of information needed to bootstrap the nodes, we can run the following two commands:

```
$ multipass exec k3snode1 -- \
   /bin/bash -c 'curl -sfL https://get.k3s.io | K3S_
URL=${K3SMASTER} K3S_TOKEN=${K3STOKEN} sh -'
$ multipass exec k3snode2 -- \
   /bin/bash -c 'curl -sfL https://get.k3s.io | K3S_
URL=${K3SMASTER} K3S_TOKEN=${K3STOKEN} sh -'
```

We should now have our three nodes configured, and all that is left is to configure our local `kubectl` so that it can interact with the cluster nodes. The first thing we need to do is copy the configuration to our local machine. To do this, run the following:

```
$ multipass exec k3smaster -- \
   /bin/bash -c 'sudo cat /etc/rancher/k3s/k3s.yaml' >
${HOME}/.kube/k3s.yml
```

If we were to use the configuration file as it is, then it would fail as, by default, K3s is configured to communicate on the localhost, so to update that run the following:

```
$ sed -ie s,https://127.0.0.1:6443,${K3SMASTER},g ${HOME}/.
kube/k3s.yml
```

As you can see, this replaces `https://127.0.0.1:6443` with the value of `${K3SMASTER}` in our local `k3s.yml` configuration file. Once replaced, we can configure `kubectl` to use our `k3s.yml` configuration file for the remainder of the Terminal session by running the following:

```
$ export KUBECONFIG=${HOME}/.kube/k3s.yml

```

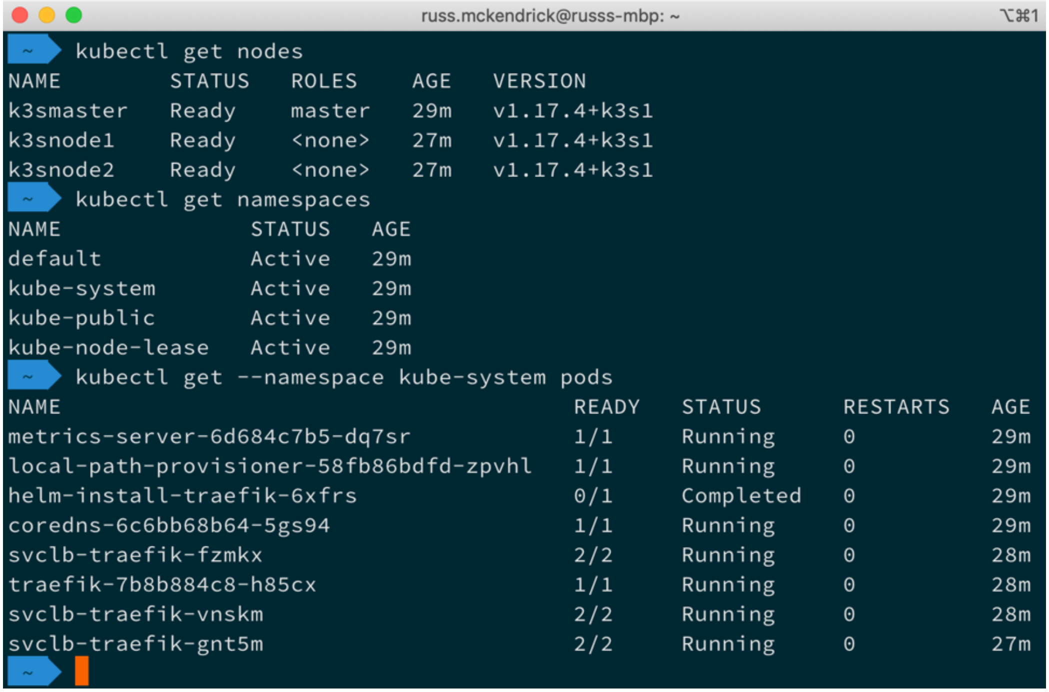
Now that our cluster is accessible using our local kubectl binary, we can launch our application.

Interacting with your Kubernetes cluster nodes

Now that we have our cluster node up and running, we can rerun the commands and launch a test application as we did in the previous section of the chapter. This time, as we have configured `kubectl` to use our `k3s.yml` configuration file, we can just run the following:

```
$ kubectl get nodes
$ kubectl get namespaces
$ kubectl get --namespace kube-system pods
```

This will give you something like the following Terminal output:



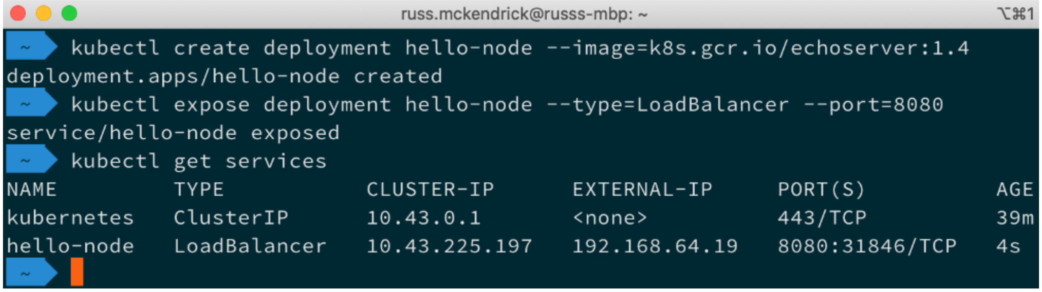
```
~ kubectl get nodes
NAME         STATUS   ROLES    AGE   VERSION
k3smaster    Ready    master   29m   v1.17.4+k3s1
k3snode1     Ready    <none>   27m   v1.17.4+k3s1
k3snode2     Ready    <none>   27m   v1.17.4+k3s1
~ kubectl get namespaces
NAME              STATUS   AGE
default           Active   29m
kube-system       Active   29m
kube-public       Active   29m
kube-node-lease   Active   29m
~ kubectl get --namespace kube-system pods
NAME                                     READY   STATUS      RESTARTS   AGE
metrics-server-6d684c7b5-dq7sr           1/1     Running     0          29m
local-path-provisioner-58fb86bdfd-zpvhl  1/1     Running     0          29m
helm-install-traefik-6xfrs               0/1     Completed   0          29m
coredns-6c6bb68b64-5gs94                 1/1     Running     0          29m
svclb-traefik-fzmkx                      2/2     Running     0          28m
traefik-7b8b884c8-h85cx                  1/1     Running     0          28m
svclb-traefik-vnskm                      2/2     Running     0          28m
svclb-traefik-gnt5m                      2/2     Running     0          27m
~
```

Figure 12.17 – Viewing the nodes, namespaces, and system pods

Next, let's launch the `hello-node` application, this time using the following commands:

```
$ kubectl create deployment hello-node --image=k8s.gcr.io/
echoserver:1.4
$ kubectl expose deployment hello-node --type=LoadBalancer
--port=8080
$ kubectl get services
```

This will give you the following output, as you can see, we have any `EXTERNAL-IP` and `PORT(S)`:



```
~ kubectl create deployment hello-node --image=k8s.gcr.io/echoserver:1.4
deployment.apps/hello-node created
~ kubectl expose deployment hello-node --type=LoadBalancer --port=8080
service/hello-node exposed
~ kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
kubernetes   ClusterIP      10.43.0.1       <none>          443/TCP          39m
hello-node   LoadBalancer   10.43.225.197   192.168.64.19   8080:31846/TCP   4s
~
```

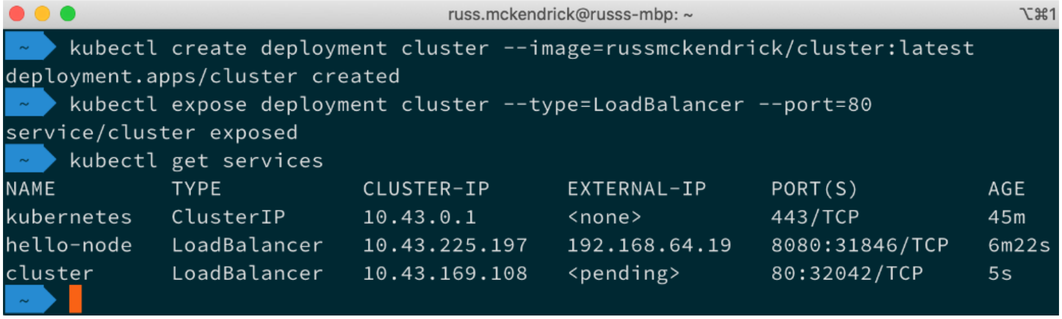
Figure 12.18 – Launching and exposing the hello-node application

Adding the IP address with the second port should give you the URL to access the application, for example, I went to `http://192.168.64.19:31846/` and was presented with the `hello-node` application.

Next up, let's launch our `cluster` application with the following commands:

```
$ kubectl create deployment cluster --image=russmckendrick/
cluster:latest
$ kubectl expose deployment cluster --type=LoadBalancer
--port=80
$ kubectl get services
```

You might notice that this time, for `EXTERNAL-IP`, it says `<pending>`:



```
~ kubectl create deployment cluster --image=russmckendrick/cluster:latest
deployment.apps/cluster created
~ kubectl expose deployment cluster --type=LoadBalancer --port=80
service/cluster exposed
~ kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
kubernetes   ClusterIP      10.43.0.1       <none>          443/TCP          45m
hello-node   LoadBalancer   10.43.225.197   192.168.64.19   8080:31846/TCP   6m22s
cluster      LoadBalancer   10.43.169.108   <pending>       80:32042/TCP     5s
~
```

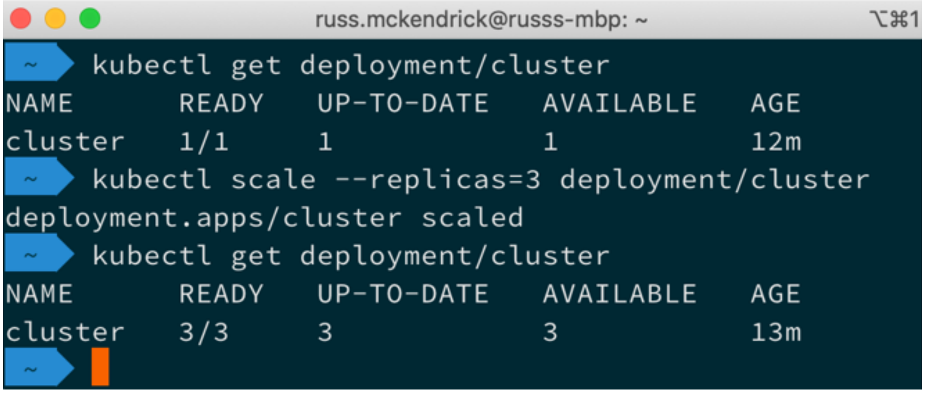
Figure 12.19 – Launching and exposing the cluster application

You shouldn't have to worry about that; just use the second port and the external IP exposed for the other server. This gave me a URL of `http://192.168.64.19:32042/` to access the `cluster` application on.

You can also scale the cluster application using the following commands:

```
$ kubectl get deployment/cluster
$ kubectl scale --replicas=3 deployment/cluster
$ kubectl get deployment/cluster
```

If you are not following along, the Terminal output for this looks like the following:



```
~ kubectl get deployment/cluster
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
cluster   1/1     1            1           12m
~ kubectl scale --replicas=3 deployment/cluster
deployment.apps/cluster scaled
~ kubectl get deployment/cluster
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
cluster   3/3     3            3           13m
~
```

Figure 12.20 – Scaling the cluster application

Before we finish up, as we have more nodes to explore, let's install the Kubernetes dashboard. To do this, run the following commands from your host machine. The first thing that is needed is to get the current version of the dashboard. To do this, run the following:

```
$ GITHUB_URL=https://github.com/kubernetes/dashboard/releases
$ VERSION_KUBE_DASHBOARD=$(curl -w '%{url_effective}' -I -L -s
-S ${GITHUB_URL}/latest -o /dev/null | sed -e 's|.*/||')
```

Now that we have the current version as an environment variable, we can run the following commands to launch the dashboard, add a user, and configure the user's access:

```
$ kubectl create -f 'https://raw.githubusercontent.com/
kubernetes/dashboard/${VERSION_KUBE_DASHBOARD}/aio/deploy/
recommended.yaml'
$ kubectl create -f 'https://raw.githubusercontent.com/
PacktPublishing/Mastering-Docker-Fourth-Edition/master/
chapter12/k3s/dashboard.admin-user.yml'
$ kubectl create -f 'https://raw.githubusercontent.com/
PacktPublishing/Mastering-Docker-Fourth-Edition/master/
chapter12/k3s/dashboard.admin-user-role.yml'
```

With the dashboard installed and the user configured, we can grab the access token we will need to log in to the dashboard by running the following:

```
$ kubectl -n kubernetes-dashboard describe secret admin-user-
token | grep ^token
```

Make a note of the token as we will need it in a second. The last thing we need to do before accessing the dashboard is to start the Kubernetes proxy service by running the following:

```
$ kubectl proxy
```

With the proxy server running, open `http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/` in your preferred browser, enter the token you made a note of, and sign in:

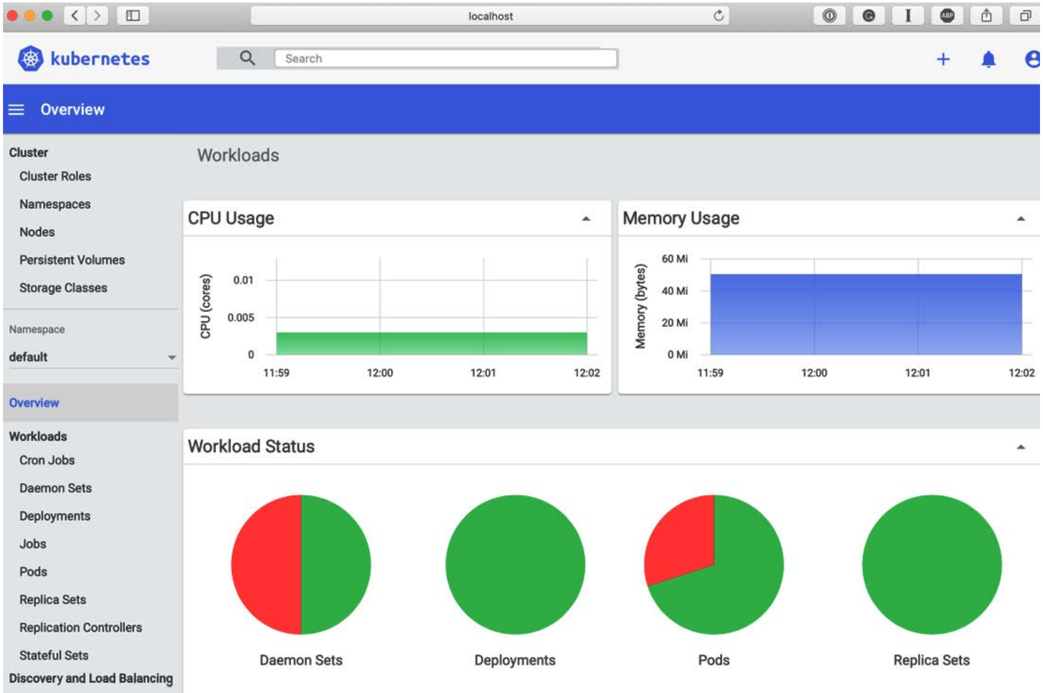


Figure 12.21 – Opening the Kubernetes dashboard on our K3s node cluster

Once you have finished with your K3s cluster, you can remove it using `the multipass delete --purge k3smaster k3snode1 k3snode2` command.

One more thing – K3d

Finally, Rancher also provides K3d. Like Kind, this is the entire Kubernetes distribution in a single container, which means that not only can you use K3s as your local development environment but it is also straightforward to introduce to your CI/CD pipelines.

Before we summarize K3s, let's take a very quick look at how you can get K3d up and running on a macOS and Linux host, starting with Linux. To install the `k3d` command, run the following:

```
$ curl -s https://raw.githubusercontent.com/rancher/k3d/master/
install.sh | bash
```

Or, if you are using macOS, you can use Homebrew to install it using the following:

```
$ brew install k3d
```

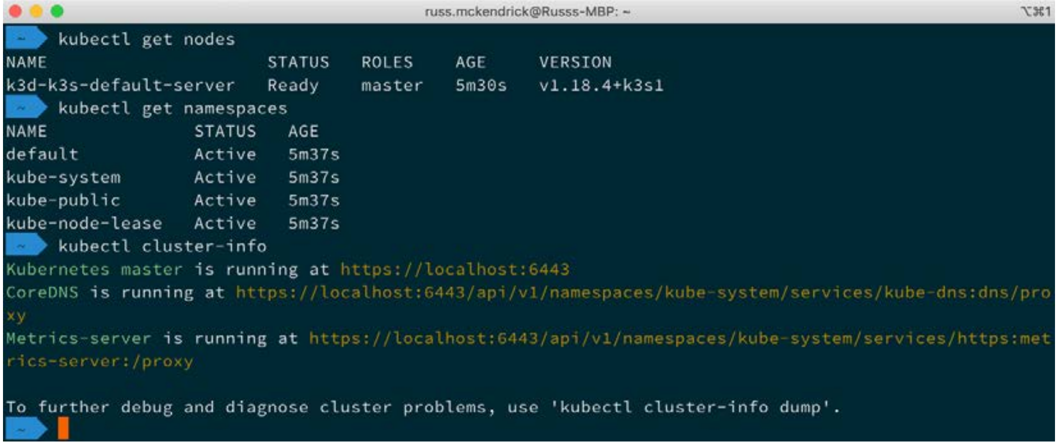
Once K3d is installed (I installed version 1.7), there are four commands we are going to look at:

- `k3d cluster create k3s-default`: This command will create a K3d-powered Kubernetes cluster called **k3s-default**.
- `k3d kubeconfig merge k3s-default --switch-context`: This will configure your local `kubectl` to talk to the **k3s-default** cluster.

Now that you have a cluster up and running, you can interact with it like any other Kubernetes cluster, for example, by running the following:

```
$ kubectl get nodes
$ kubectl get namespaces
$ kubectl cluster-info
```

This gives us the following output:



```
~ kubectl get nodes
NAME                     STATUS   ROLES    AGE     VERSION
k3d-k3s-default-server   Ready    master   5m30s   v1.18.4+k3s1
~ kubectl get namespaces
NAME              STATUS   AGE
default           Active   5m37s
kube-system       Active   5m37s
kube-public       Active   5m37s
kube-node-lease   Active   5m37s
~ kubectl cluster-info
Kubernetes master is running at https://localhost:6443
CoreDNS is running at https://localhost:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/pro
xy
Metrics-server is running at https://localhost:6443/api/v1/namespaces/kube-system/services/https:met
rics-server:/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
~
```

Figure 12.22 – Running commands against our K3d cluster

- `k3dcluster delete k3s-default` is the final command of the four we are going to look at, and as you might have guessed, this deletes the cluster.

K3d is very much in active development. In fact, a complete rewrite of the K3d wrapper has just been completed so if the preceding commands, which cover the new version 3, do not work, then try the following commands, which cover the old version:

```
$ k3d create
$ export KUBECONFIG='$(k3d get-kubeconfig --name='k3s-
default')': This will configure your local kubectl to talk to
the k3s-default cluster.
$ k3d delete
```

For more up-to-date news on the development of K3d, see the project's GitHub page, which is linked in the *Further reading* section of this chapter.

K3s summary

Like MicroK8s, K3s delivers on its promise of a lightweight Kubernetes distribution.

Personally, I find K3s to be the better of the two as it feels more like a fully formed Kubernetes distribution than MicroK8s does.

The other thing K3s has going for it is that deploying a local multi-node cluster is a relatively painless experience. This should give your local development environment a much more production-like feeling, and while Minikube does allow you to launch a multi-node cluster, the functionality is still in its infancy and is not really ready for public consumption yet.

Summary

In this chapter, we looked at four different tools for launching both single-node and multi-node Kubernetes clusters. We discovered that while the method of launching each of the clusters is slightly different, once they are up and running, you get a mostly consistent experience once you start to interact with them using standard Kubernetes tools like `kubectl`.

At this point, I should probably confess something: two of the four tools we have covered in this chapter do not actually use Docker in the traditional sense – both MicroK8s and K3s actually use **containerd**.

As you may recall from *Chapter 1*, *Docker Overview*, **containerd** is an easily embeddable container runtime. It started life at Docker Inc., but the project was donated to the **Cloud Native Computing Foundation** (**CNCF**) – it is the container runtime of the Moby project, which Docker uses as its upstream project.

It is not only small and lightweight, but it also offers full OCI Image and OCI Runtime specification support, meaning that it is 100% compatible with Docker images and also the way that Docker runs containers.

In the next chapter, we are going to move away from running Kubernetes locally and take our clusters to the cloud.

Questions

1. True or false: Kind is recommended for production use.
2. Name at least two tools we have looked at in this chapter that are able to run inside a Docker container.
3. If you had an ARM-powered IoT device, which two Kubernetes distributions that we have covered in this chapter could you use?

Further reading

Some of the Google tools, presentations, and white papers mentioned at the start of the chapter can be found here:

- Minikube: `https://minikube.sigs.k8s.io/`
- Kind: `https://kind.sigs.k8s.io/`
- MicroK8s: `https://microk8s.io/`
- K3s: `https://k3s.io/`
- K3d: `https://github.com/rancher/k3d`
- containerd: `https://containerd.io/`
- OCI Image and Runtime Specification: `https://www.opencontainers.org/`
- Certified Kubernetes offerings: `https://www.cncf.io/certification/software-conformance/`

13

Running Kubernetes in Public Clouds

In the previous two chapters, we looked at the various ways that we can create, configure, and interact with Kubernetes clusters on our local machines.

Now it is time to take our Kubernetes journey into the cloud by looking at what is involved in launching, configuring, and using Kubernetes in the following public cloud providers using both the web portals from each provider and also their command-line tools:

- **Microsoft Azure Kubernetes Service** (**AKS**)
- **Google Kubernetes Engine** (**GKE**)
- **Amazon Elastic Kubernetes Service** (**EKS**)
- **DigitalOcean Kubernetes**

Technical requirements

This chapter assumes that you have access to one or all of the cloud providers that we will be covering. As such, please be aware that launching these services, depending on your account, will incur costs. You should also remember to remove any resources once you have finished with them or that you understand the costs if you choose to leave them running for any length of time.

As with the previous chapters, I will be using my preferred operating system, which is macOS. As such, some of the supporting commands, which will be few and far between, may only apply to macOS.

Check out the following video to see the Code in Action:
`https://bit.ly/336J7dE`

Microsoft Azure Kubernetes Service (AKS)

Microsoft has long been a supporter of running container workloads in Microsoft Azure. Originally, Microsoft started by offering the Azure Container Service, which supported three different container orchestrators: Kubernetes, Mesosphere DC/OS, and Docker Swarm.

However, in October 2017, Microsoft announced that they would be replacing Azure Container Service with the newly developed Azure Kubernetes Service—this, as you may already have guessed, dropped support for Mesosphere DC/OS and Docker Swarm.

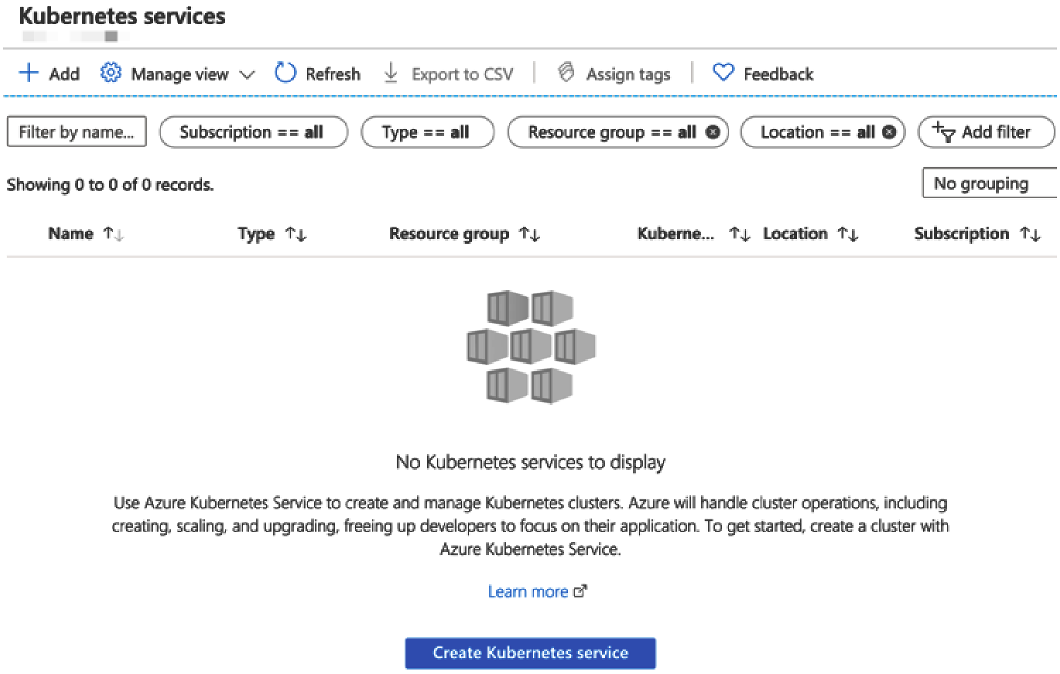
Since then, the service, which is a CNCF-certified Kubernetes hosting platform, has come on leaps and bounds, with a recent development being the general availability of Windows container support.

Rather than go into any more detail on this subject, let's get on and launch an AKS cluster. We will be covering two ways of doing this: using the Azure web portal and setting it up from your local machine using the Azure command-line tools. Let's start with the Azure web portal.

Launching a cluster using the web portal

To start with, you need to be logged into the Azure web portal, which can be found at `https://portal.azure.com/`. Once logged in, enter `AKS` in the top search bar and click on the **Kubernetes services** item in the **Services** section of the search results.

Once the page has loaded, you will be presented with a screen that looks like the following:



Showing 0 to 0 of 0 records.

No Kubernetes services to display

Use Azure Kubernetes Service to create and manage Kubernetes clusters. Azure will handle cluster operations, including creating, scaling, and upgrading, freeing up developers to focus on their application. To get started, create a cluster with Azure Kubernetes Service.

Figure 13.1: Ready to create the cluster

Most of you will have already guessed what the next step in launching our AKS cluster is going to be. That's right: click on the + **Add** button and select **Add Kubernetes cluster**. There are seven sections that we are going to be working through to launch our cluster, listed as follows:

- **Basics**
- **Node pools**
- **Authentication**
- **Networking**
- **Integrations**
- **Tags**
- **Review + create**

Let's start with the **Basics**. Here, you will be asked for several pieces of information, beginning with the **Project details**. Here, we need to define which **Subscription** and **Resource Group** should be used.

Choose your required subscription and then click on the **Create new** link under **Resource Group**. Enter the name of the resource group you would like your cluster to be placed—I have called mine **mastering-docker-aks-rg**.

Once those two bits of information have been entered, we can move on to the **Cluster details**. In this section, we need to provide the **Kubernetes cluster name**, for which I entered **mastering-docker-aks**. You should also enter the Azure **Region** that you would like the cluster to be launched in—I used **(US) East US** . Lastly, you should enter the **Kubernetes version**, which I left as the default (`1.15.10` at the time of writing).

The last piece of basic information is for the **Primary node pool**. Here, we can define the **Node size** and the **Node count**. By default, these are set to **Standard DS2 v2** and **3**—for testing purposes, let's leave these as their default settings. To progress to the next step, click on the **Next: Node pools >** button.

On the **Node pools** screen, we can add more pools—for example, if you wanted to run a Windows host pool alongside the Linux one that we are launching, then you can click on the **+ Add node pool** button.

Here, you will be asked the following:

- **Node pool name**: Name the new node pool.
- **OS type**: The underlying operating system that you want the new node pool to use. You have the choice of Linux or Windows.
- **Node size**: The virtual machine size that the nodes in the new node pool should be.
- **Node count**: The number of nodes that you want in the pool.

As we are launching a test cluster, we do not need any additional node pools, so let's move on to the next set of options.

Virtual nodes allow you to use **Azure Container Instances**, which we looked at in *Chapter 10*, *Running Docker in Public Clouds*. Leave this setting as **Disabled**.

Setting **VM scale sets** as **Enabled** will allow us to scale our cluster up and down, so let's click on the **Next: Authentication** > button.

The **Authentication** settings allow you to configure how the Kubernetes cluster will be able to interact with and even launch other Azure services. For our cluster, leave these at their defaults and click on the **Next: Networking** > button.

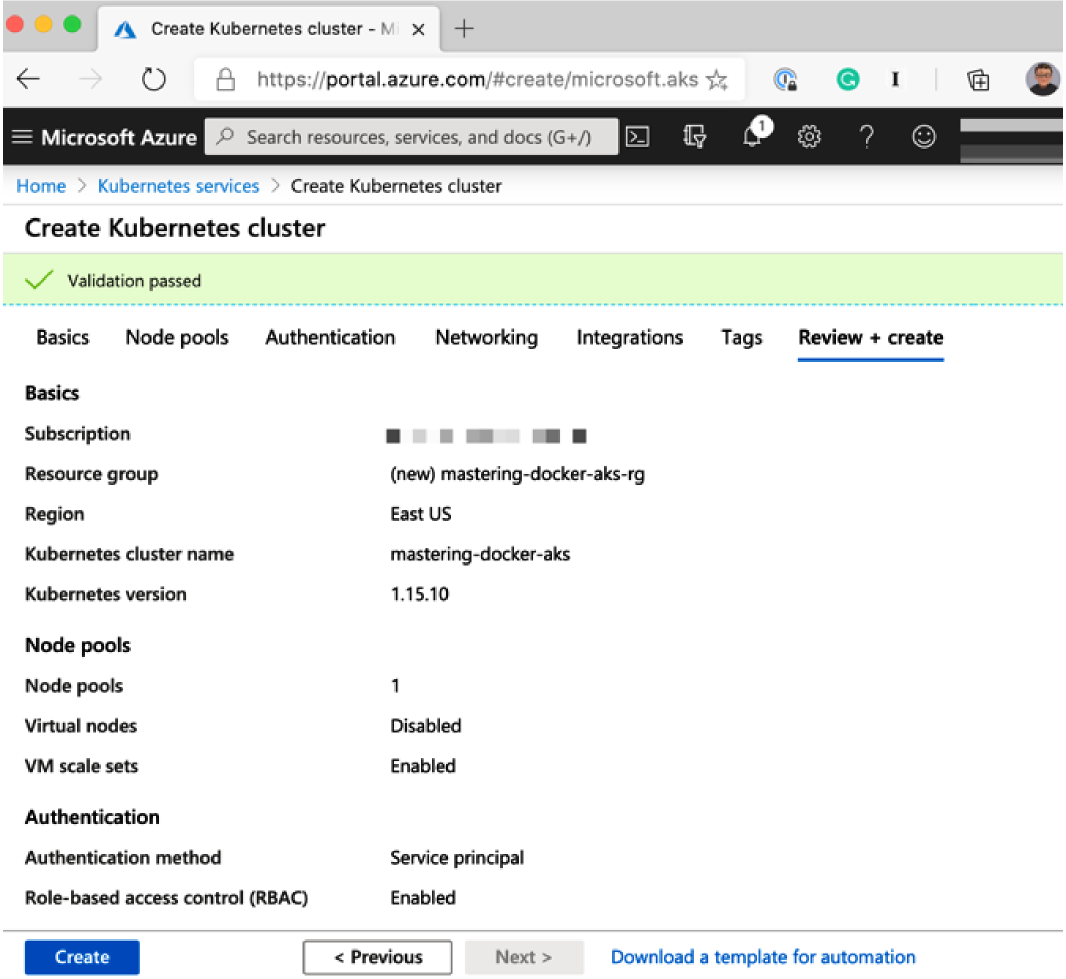
The **Networking** page has two different sets of options, the first being the **Network Configuration**. Here, we have the option of choosing an existing **Virtual network** or creating one—for ease of use, let's stick with the defaults, which are already prefilled in for us, and move on to the **Network Settings**.

Here, we can again stick with the defaults, meaning that the only piece of information we need to provide is the **DNS name prefix**—I used **mastering-docker-aks**.

Clicking on **Next: Integrations** > will take you to the options, where you can connect your AKS cluster to an **Azure Container Registry**, which we discussed in *Chapter 3, Storing & Distributing Images*. But as we are not using a system-assigned managed identity, this option is not available. The other integration we can configure is **Azure Monitor**. Leave these settings at their defaults, as we want a **Log Analytics workspace** set up for us.

As our cluster is only temporary, you can click on the **Review + Create** button rather than adding the **Tags**.

The first thing that happens is that your configuration is validated. If there are any problems, you will not be allowed to proceed until the problem is fixed. If everything goes as planned, then you should be presented with something that looks like the following screen:


Create Kubernetes cluster - M
https://portal.azure.com/#create/microsoft.aks
Microsoft Azure
Search resources, services, and docs (G+/)
Home > Kubernetes services > Create Kubernetes cluster

Create Kubernetes cluster

✓ Validation passed

Basics
Node pools
Authentication
Networking
Integrations
Tags
Review + create

Basics

Subscription [redacted]
Resource group (new) mastering-docker-aks-rg
Region East US
Kubernetes cluster name mastering-docker-aks
Kubernetes version 1.15.10

Node pools

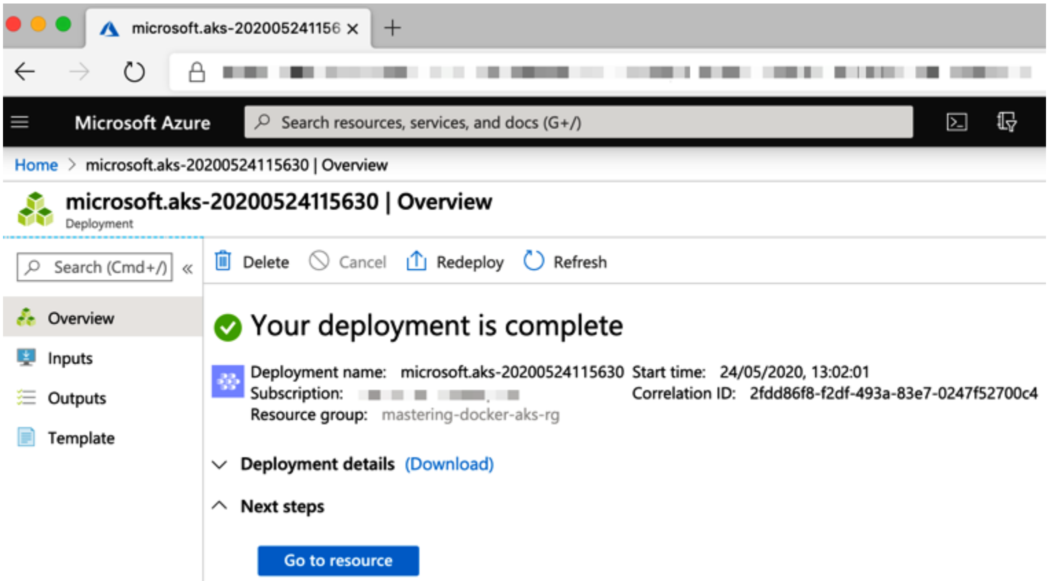
Node pools 1
Virtual nodes Disabled
VM scale sets Enabled

Authentication

Authentication method Service principal
Role-based access control (RBAC) Enabled

Figure 13.2: A validated configuration

Once validated, click on the **Create** button. Now would be a good time to grab a drink as a cluster could take anywhere from 10 to 20 minutes to deploy. Once deployed, you will see the following screen:


microsoft.aks-202005241156
[redacted]
Microsoft Azure
Search resources, services, and docs (G+/)
Home > microsoft.aks-20200524115630 | Overview
Search (Cmd+/)
Overview
Inputs
Outputs
Template
Delete
Cancel
Redeploy
Refresh
Deployment details (Download)
Next steps
Go to resource

microsoft.aks-20200524115630 | Overview
Deployment

Your deployment is complete

Deployment name: microsoft.aks-20200524115630
Subscription: [redacted]
Resource group: mastering-docker-aks-rg
Start time: 24/05/2020, 13:02:01
Correlation ID: 2fdd86f8-f2df-493a-83e7-0247f52700c4

Figure 13.3: The deployment has completed

Clicking on **Go to resource** will take you to your cluster. The final part of the setup is to configure the newly created cluster to a `kubectl` instance.

As Microsoft provides a shell built into Azure portal with both the Azure command-line tools and kubectl preinstalled; because we launched the cluster through the portal, we will use that.

Click on the Cloud Shell icon, which is the first of the icons in the top right of the screen after the search bar (it has >_ on it). If you have used Cloud Shell before, then you will be logged straight in. If not, then follow the on-screen prompts.

There are two different flavors of Cloud Shell: **Bash** and **PowerShell**. For our purposes, you need to make sure you are using **Bash**.

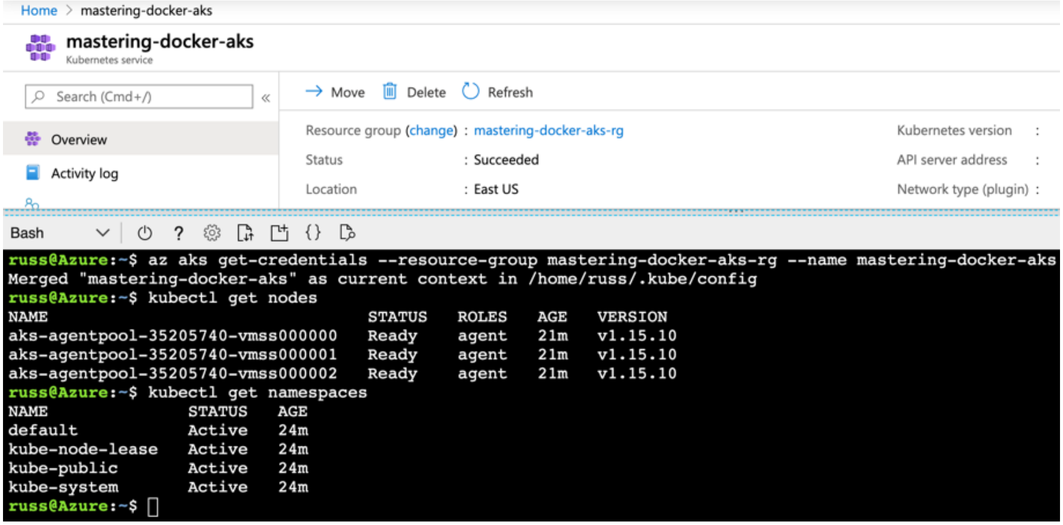
Once you are at Command Prompt, enter the following command, making sure that you update the resource group and cluster name if you have used different values to the ones I used:

```
$ az aks get-credentials --resource-group mastering-docker-
aks-rg --name mastering-docker-aks
```

Once you have run the command, you can test the connectivity to the cluster by running the following:

```
$ kubectl get nodes
$ kubectl get namespaces
```

This should return three nodes, each with the version of Kubernetes that we selected when we first went to configure our cluster:



Resource group (change) : mastering-docker-aks-rg
Status : Succeeded
Location : East US
Kubernetes version :
API server address :
Network type (plugin) :

```
russ@Azure:~$ az aks get-credentials --resource-group mastering-docker-aks-rg --name mastering-docker-aks
Merged "mastering-docker-aks" as current context in /home/russ/.kube/config
russ@Azure:~$ kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-agentpool-35205740-vmss000000   Ready    agent   21m   v1.15.10
aks-agentpool-35205740-vmss000001   Ready    agent   21m   v1.15.10
aks-agentpool-35205740-vmss000002   Ready    agent   21m   v1.15.10
russ@Azure:~$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   24m
kube-node-lease   Active   24m
kube-public       Active   24m
kube-system       Active   24m
russ@Azure:~$ 
```

Figure 13.4: Connecting kubectl to our AKS cluster using Cloud Shell

Now we have our cluster up and running, you can skip the next part of this section where we launch the cluster using the Azure command-line tools and go straight to launching an application.

Launching a cluster using the command-line tools

An alternative to launching the cluster through the Azure web portal is to use the Azure command-line tool. If you don't already have it installed, then you can use the following instructions, starting with macOS:

```
$ brew install azure-cli
```

Windows users can open a PowerShell prompt as the Administrator user and run the following:

```
$ Invoke-WebRequest -Uri https://aka.ms/installazurecliwindows
 -OutFile .\AzureCLI.msi; Start-Process msiexec.exe -Wait
-ArgumentList '/I AzureCLI.msi /quiet'; rm .\AzureCLI.msi
```

Linux users using Debian-based machines can run the following:

```
$ curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

Linux users running Red-Hat-based operating systems can run the following:

```
$ sudo rpm --import https://packages.microsoft.com/keys/
microsoft.asc
$ sudo sh -c 'echo -e '[azure-cli]
name=Azure CLI
baseurl=https://packages.microsoft.com/yumrepos/azure-cli
enabled=1
gpgcheck=1
gpgkey=https://packages.microsoft.com/keys/microsoft.asc' > /
etc/yum.repos.d/azure-cli.repo'
$ sudo yum install azure-cli
```

Once you have the Azure CLI package installed, the first command you should run is the following:

```
$ az login
```

This will prompt you to log in to your Azure account.

Important note

Please make sure that the account you are logging in to has the subscription that you want to launch your resources in as the default subscription—if it doesn't, go to `https://docs.microsoft.com/en-us/cli/azure/manage-azure-subscriptions-azure-cli` for details on how to change the active subscription.

Now that we have the Azure command-line tools installed and configured, we can launch our cluster. For macOS and Linux users, we can set some environment variables for values that we will be using through the cluster creation.

Important note

Windows users should replace the variables in the commands used to launch the cluster with the corresponding values from the following four variables.

These are the same values that we used in the last section, where we launched our cluster using the Azure web portal:

```
$ AKSLOCATION=eastus
$ AKSRG=mastering-docker-aks-rg
$ AKSCLUSTER=mastering-docker-aks
$ AKSNUMNODES=3
```

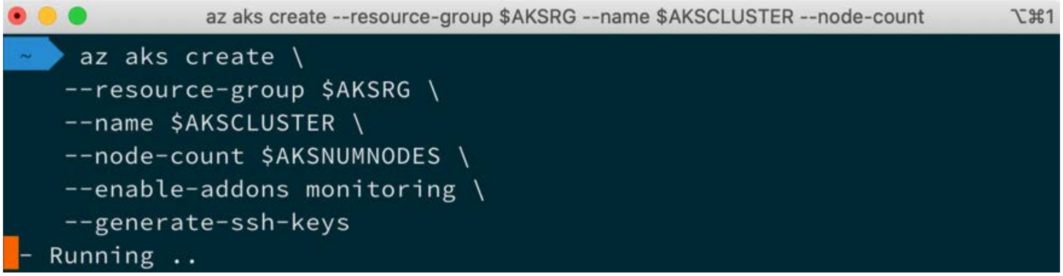
The first step in launching the cluster is to create the resource group. To do this, run the following:

```
$ az group create --name $AKSRG --location $AKSLOCATION
```

This should return a JSON file where the provisioning state should be `Succeeded`. Once the resource group has been created, we can launch our cluster by running the following command:

```
$ az aks create \
      --resource-group $AKSRG \
      --name $AKSCLUSTER \
      --node-count $AKSNUMNODES \
      --enable-addons monitoring \
      --generate-ssh-keys
```

While the cluster is launching, you will see a screen that looks like the following:



```
az aks create --resource-group $AKSRG --name $AKSCLUSTER --node-count    ⌥⌘1
~  az aks create \
  --resource-group $AKSRG \
  --name $AKSCLUSTER \
  --node-count $AKSNUMNODES \
  --enable-addons monitoring \
  --generate-ssh-keys
- Running ..
```

Figure 13.5: Launching the cluster using the Azure CLI

Just like when the cluster was launched using the Azure web portal, it will take between 10 and 20 minutes to complete, so don't worry too much if you see the preceding output for a while—it just takes a little time.

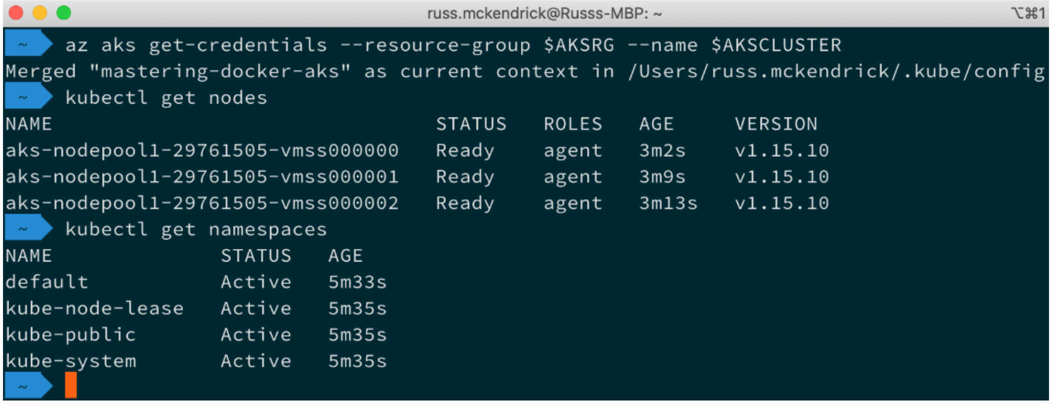
Once completed, the final step is the same as when we launched the cluster using the Azure web portal. We need to configure kubectl to point towards the new launching cluster. To do this, run the following command:

```
$ az aks get-credentials --resource-group $AKSRG --name
$AKSCLUSTER
```

Once the configuration has been merged, you can run the following commands to check the connectivity with the cluster:

```
$ kubectl get nodes
$ kubectl get namespaces
```

This should return something similar to the following output:



```
~ az aks get-credentials --resource-group $AKSRG --name $AKSCLUSTER
Merged "mastering-docker-aks" as current context in /Users/russ.mckendrick/.kube/config
~ kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-nodepool1-29761505-vmss000000   Ready    agent   3m2s    v1.15.10
aks-nodepool1-29761505-vmss000001   Ready    agent   3m9s    v1.15.10
aks-nodepool1-29761505-vmss000002   Ready    agent   3m13s   v1.15.10
~ kubectl get namespaces
NAME              STATUS   AGE
default           Active   5m33s
kube-node-lease   Active   5m35s
kube-public       Active   5m35s
kube-system       Active   5m35s
~
```

Figure 13.6: Checking the cluster connectivity using kubectl

We should now have an AKS cluster in the same state as if it were launched using the Azure web portal, which means that we can now launch an application.

Launching an application

As we are no longer using our local machine to run Kubernetes, we can launch an application that is a little more resource-intensive. For this, we are going to be using the microservices demo created and maintained by Weave.

To launch the application, you simply need to run the following commands:

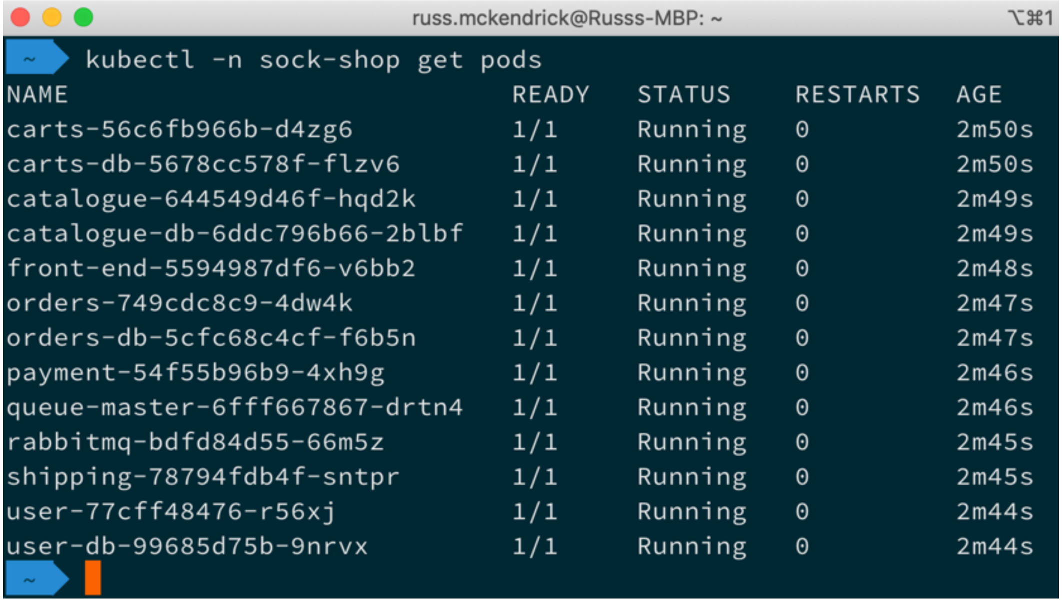
```
$ kubectl create namespace sock-shop
$ kubectl -n sock-shop apply -f 'https://github.com/
microservices-demo/microservices-demo/blob/master/deploy/
kubernetes/complete-demo.yaml?raw=true'
```

This will create a namespace called **sock-shop** and then the resource that is used to make up the application, which is an e-commerce site selling socks.

After a minute or two, you can run the following command to view the status of the pods:

```
$ kubectl -n sock-shop get pods
```

This should display something like the following output:



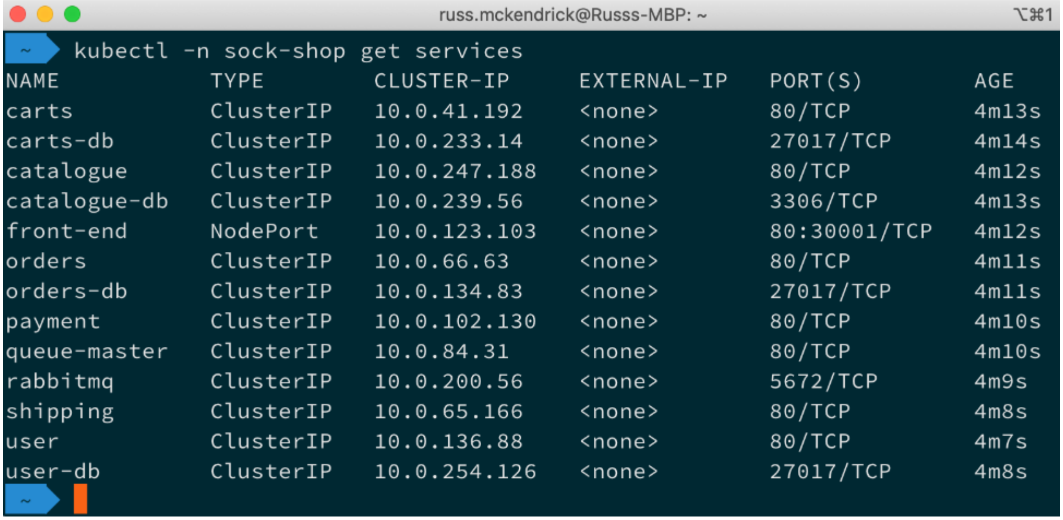
```
~  kubectl -n sock-shop get pods
NAME                             READY   STATUS    RESTARTS   AGE
carts-56c6fb966b-d4zg6           1/1     Running   0          2m50s
carts-db-5678cc578f-flzv6        1/1     Running   0          2m50s
catalogue-644549d46f-hqd2k       1/1     Running   0          2m49s
catalogue-db-6ddc796b66-2blbf    1/1     Running   0          2m49s
front-end-5594987df6-v6bb2       1/1     Running   0          2m48s
orders-749cdc8c9-4dw4k           1/1     Running   0          2m47s
orders-db-5cfc68c4cf-f6b5n       1/1     Running   0          2m47s
payment-54f55b96b9-4xh9g         1/1     Running   0          2m46s
queue-master-6fff667867-drtn4    1/1     Running   0          2m46s
rabbitmq-bdfd84d55-66m5z         1/1     Running   0          2m45s
shipping-78794fdb4f-sntpr        1/1     Running   0          2m45s
user-77cff48476-r56xj            1/1     Running   0          2m44s
user-db-99685d75b-9nrvx          1/1     Running   0          2m44s
~
```

Figure 13.7: Checking the status of the pods

Once all of the pods are running, you can check the status of the services by running the following:

```
$ kubectl -n sock-shop get services
```

Again, you should see something like the following:



```
~ kubectl -n sock-shop get services
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
carts          ClusterIP   10.0.41.192    <none>        80/TCP         4m13s
carts-db       ClusterIP   10.0.233.14    <none>        27017/TCP      4m14s
catalogue      ClusterIP   10.0.247.188   <none>        80/TCP         4m12s
catalogue-db   ClusterIP   10.0.239.56    <none>        3306/TCP       4m13s
front-end      NodePort    10.0.123.103   <none>        80:30001/TCP   4m12s
orders         ClusterIP   10.0.66.63     <none>        80/TCP         4m11s
orders-db      ClusterIP   10.0.134.83    <none>        27017/TCP      4m11s
payment        ClusterIP   10.0.102.130   <none>        80/TCP         4m10s
queue-master   ClusterIP   10.0.84.31     <none>        80/TCP         4m10s
rabbitmq       ClusterIP   10.0.200.56    <none>        5672/TCP       4m9s
shipping       ClusterIP   10.0.65.166    <none>        80/TCP         4m8s
user           ClusterIP   10.0.136.88    <none>        80/TCP         4m7s
user-db        ClusterIP   10.0.254.126   <none>        27017/TCP      4m8s
~
```

Figure 13.8: Checking the status of the services

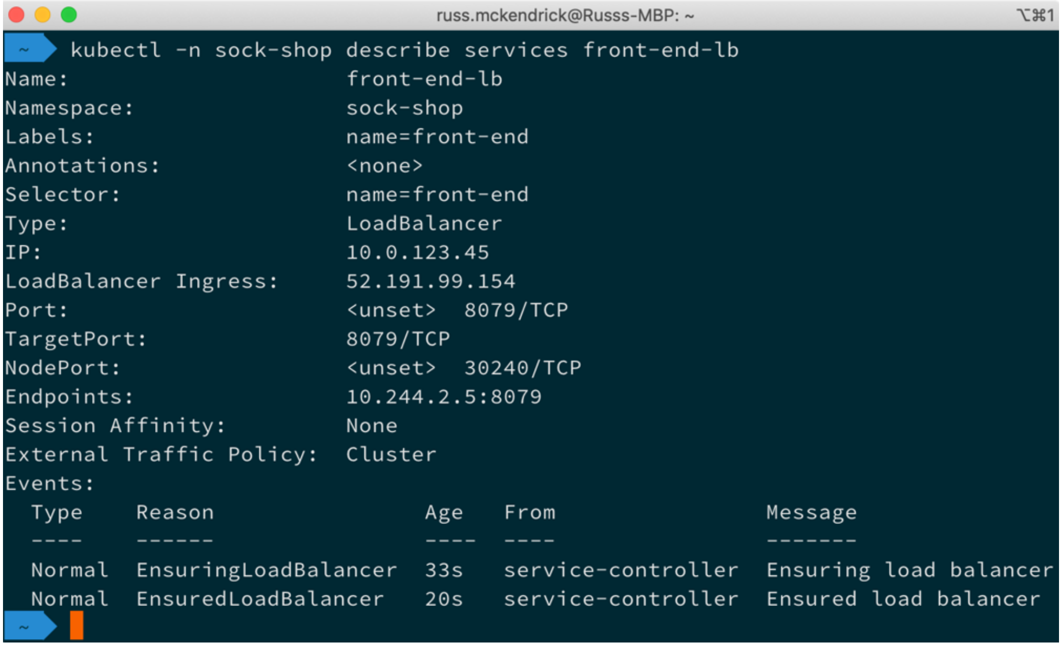
Now that the application has been deployed, we need to expose it so that we can access it. To do this, run the following command:

```
$ kubectl -n sock-shop expose deployment front-end
--type=LoadBalancer --name=front-end-lb
```

Once exposed, you can run the following command to get information on the endpoint:

```
$ kubectl -n sock-shop describe services front-end-lb
```

In the output, you are after the `LoadBalancer Ingress` and `Port`. See the following screenshot for an example output:



```
~  kubectl -n sock-shop describe services front-end-lb
Name:                     front-end-lb
Namespace:                sock-shop
Labels:                   name=front-end
Annotations:              <none>
Selector:                 name=front-end
Type:                     LoadBalancer
IP:                       10.0.123.45
LoadBalancer Ingress:     52.191.99.154
Port:                     <unset>  8079/TCP
TargetPort:               8079/TCP
NodePort:                 <unset>  30240/TCP
Endpoints:                10.244.2.5:8079
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason                Age   From                Message
  ----    ------                ----  ----                -------
  Normal  EnsuringLoadBalancer  33s   service-controller  Ensuring load balancer
  Normal  EnsuredLoadBalancer   20s   service-controller  Ensured load balancer
~
```

Figure 13.9: Getting information on the exposed service

Entering the `LoadBalancer Ingress` IP and `Port`—which for me was `http://52.191.99.154:8079/`—into a browser should open the store:

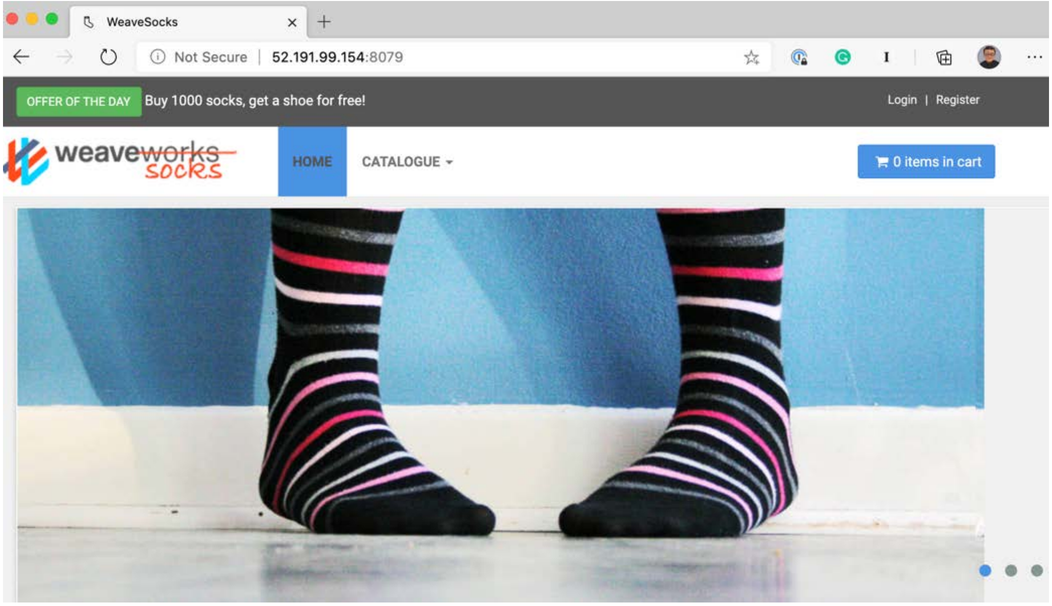
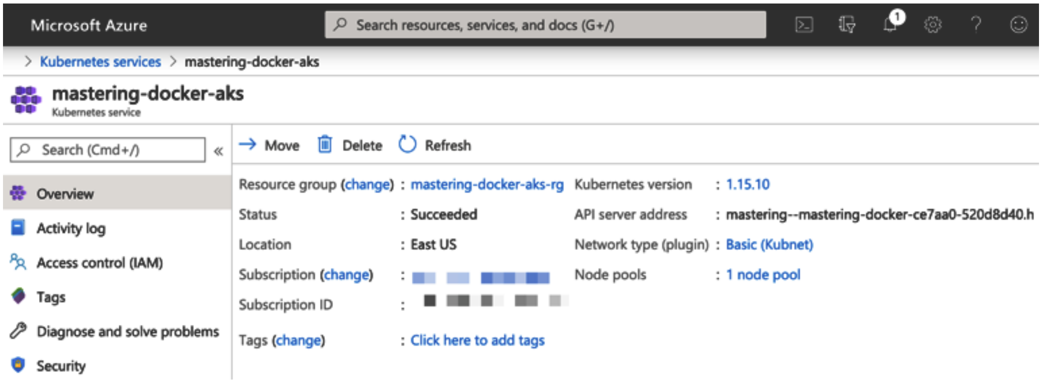


Figure 13.10 – The Sock Shop application

Click around the application—stick some socks in your basket, refresh pages, and so on. We are now going to take a look at the cluster in the Azure web portal.

Cluster information

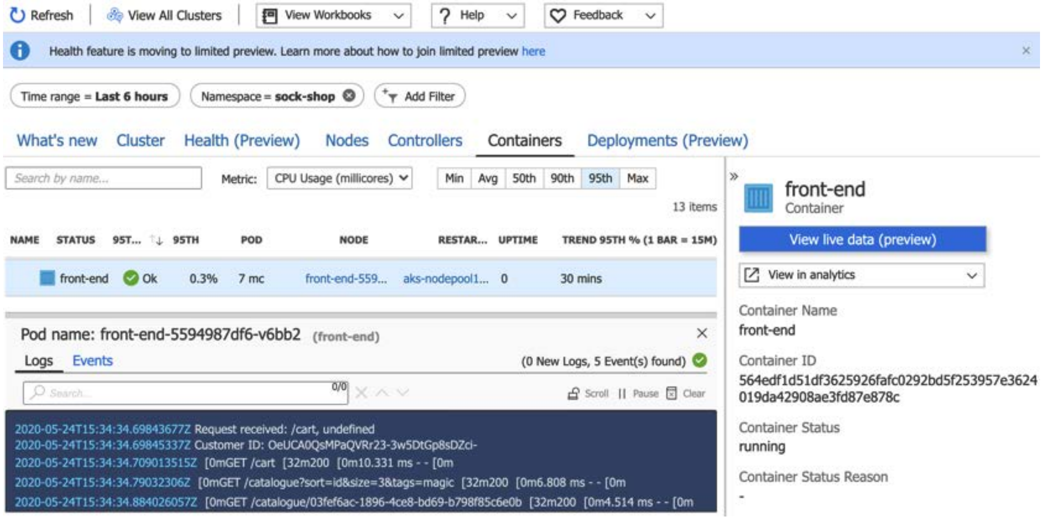
If you log in to the Azure web portal at `https://portal.azure.com/` and then search for the name of your cluster in the search bar at the top of the page, you should be greeted with a page that looks like the following:



Resource group (change) : mastering-docker-aks-rg
Status : Succeeded
Location : East US
Subscription (change) : [redacted]
Subscription ID : [redacted]
Tags (change) : Click here to add tags
Kubernetes version : 1.15.10
API server address : mastering--mastering-docker-ce7aa0-520d8d40.h
Network type (plugin) : Basic (Kubnet)
Node pools : 1 node pool

Figure 13.11 – Viewing the cluster in the Azure web portal

If you then click on the **Monitor container** button, you will be taken to a **Cluster** overview. Once there, click on **Containers**. Adding a filter for the **sock-shop** namespace will just select the containers that are running in the pods for our application. Selecting one and then clicking the **View live data (preview)** button will stream the logs from the container to the page:



Time range = **Last 6 hours**
Namespace = **sock-shop**
Metric: CPU Usage (millicores)
13 items

NAME	STATUS	95T...	95TH	POD	NODE	RESTAR...	UPTIME	TREND 95TH % (1 BAR = 15M)
front-end	Ok	0.3%	7 mc	front-end-559...	aks-nodepool1...	0	30 mins	

Pod name: front-end-5594987df6-v6bb2 (front-end)
(0 New Logs, 5 Event(s) found)

```
2020-05-24T15:34:34.69843677Z Request received: /cart, undefined
2020-05-24T15:34:34.69845337Z Customer ID: OeUCA0QsMPaQVRr23-3w5DtGp8sDZci-
2020-05-24T15:34:34.709013515Z  [0mGET /cart  [32m200  [0m10.331 ms - - [0m
2020-05-24T15:34:34.79032306Z  [0mGET /catalogue?sort=id&size=3&tags=magic  [32m200  [0m6.808 ms - - [0m
2020-05-24T15:34:34.884026057Z  [0mGET /catalogue/03fef6ac-1896-4ce8-bd69-b798f85c6e0b  [32m200  [0m4.514 ms - - [0m
```

front-end
Container

Container Name
front-end

Container ID
564edf1d51df3625926fafc0292bd5f253957e3624019da42908ae3fd87e878c

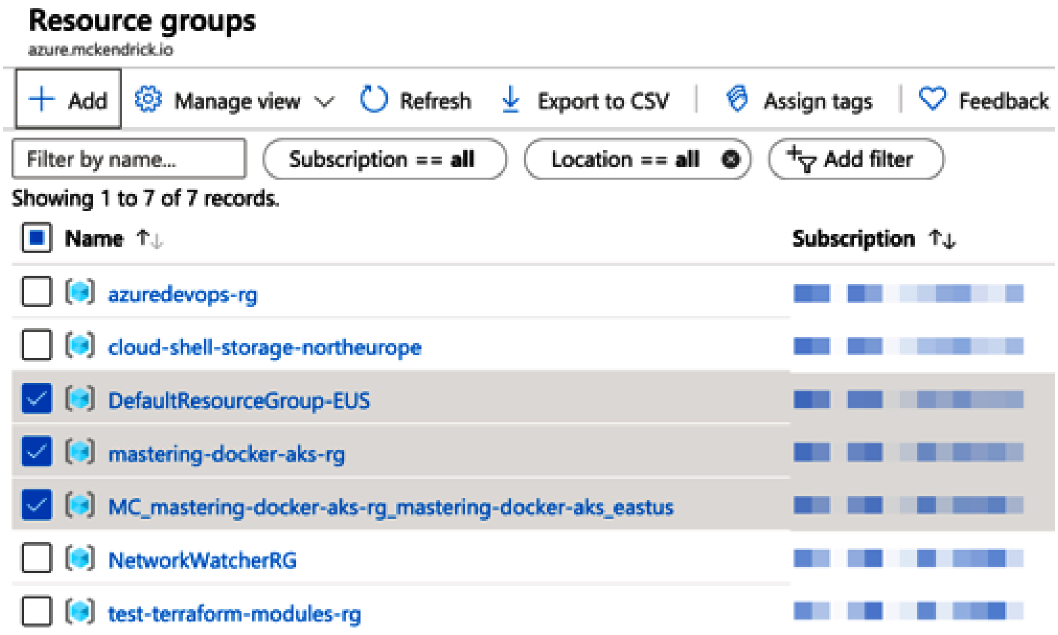
Container Status
running

Container Status Reason
-

Figure 13.12 – Viewing the logs from the frontend container in the Azure web portal

I would recommend having a click around some of the other tabs and options to get an idea of what else you can do with AKS.

Once you have finished with your AKS cluster, I recommend removing all of the resources created either through the Azure web portal or Azure CLI by deleting the resource groups. To do this, enter `Resource Groups` in the top search box and select the resource group related to your cluster. For example, as I launched my cluster, I have three resource groups, which I have highlighted in the following screenshot:



Showing 1 to 7 of 7 records.

	Name ↑↓	Subscription ↑↓
☐	azuredevops-rg	[redacted]
☐	cloud-shell-storage-northeurope	[redacted]
☑	DefaultResourceGroup-EUS	[redacted]
☑	mastering-docker-aks-rg	[redacted]
☑	MC_mastering-docker-aks-rg_mastering-docker-aks_eastus	[redacted]
☐	NetworkWatcherRG	[redacted]
☐	test-terraform-modules-rg	[redacted]

Figure 13.13 – Finding the resource groups to delete

To remove them, click on the resource group name, double-check the resources in there and then if you are happy, they contain just the resources for your test cluster click on the **Delete resource group** button and follow the on-screen instructions.

Microsoft Azure Kubernetes Service summary

As I am sure you will agree, launching the cluster was quite a straightforward process using both the Azure web portal and Azure CLI. Once launched and kubectl configured to interact with the cluster, the commands that were used to launch the application were pretty much the same as the ones that we used in *Chapter 12, Discovering other Kubernetes options*.

The cluster that we launched would have a running cost of around $215 per month, with the only cost being for the virtual machine resource. Microsoft does not charge for the resources required for cluster management; if we want to add an uptime SLA to the cluster, we would add a further $73.00 per month.

Being able to run multiple node pools with a mixture of Linux and Windows containers is a big plus point for the service, as is its integration with Azure Monitor and the Azure Container Registry service. Add to this services such as Azure DevOps and **Azure Sentinel for Security information and event management** (**SIEM**), and you have quite a powerful platform.

Next up, we are going to move from Microsoft Azure to Google Cloud.

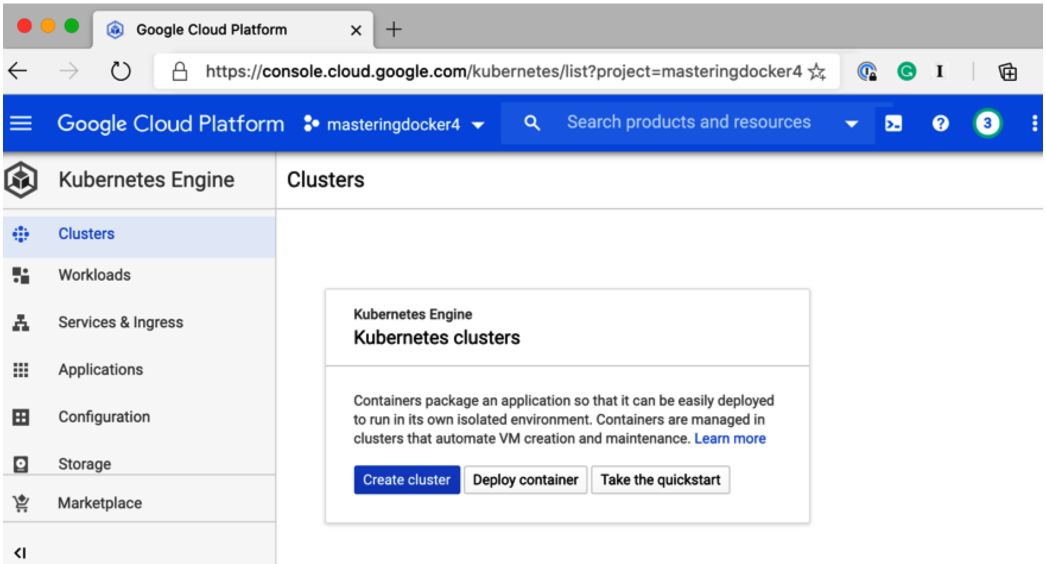
Google Kubernetes Engine (GKE)

The GKE, as you may have already guessed, is very tightly integrated with Google's Cloud platform. Rather than going into more detail on how Kubernetes started off life at Google, let's dive straight in and launch a cluster.

Going ahead, I am assuming that you already have a Google Cloud account and a project with billing enabled.

Launching a cluster using the web portal

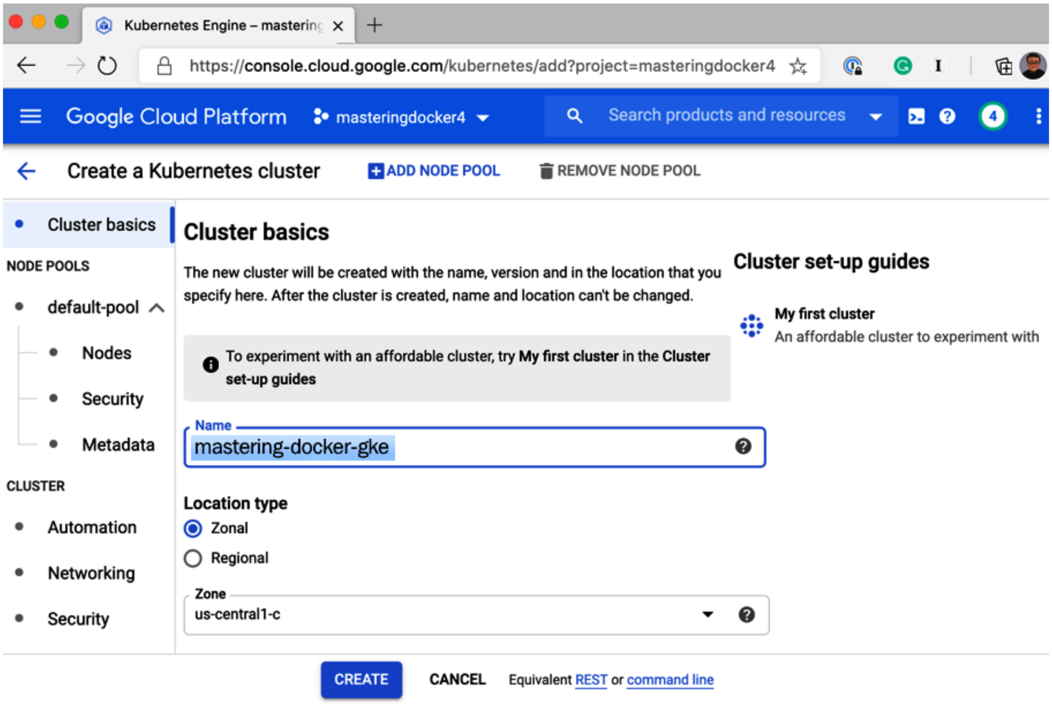
Once you are logged into `https://console.cloud.google.com/`, enter Kubernetes into the search box at the top of the page and select **Kubernetes Engine**. If you don't have the service enabled for your project, it will automatically be enabled, and after a few seconds, you will be presented with a page that looks like the following:



Containers package an application so that it can be easily deployed to run in its own isolated environment. Containers are managed in clusters that automate VM creation and maintenance. Learn more

Figure 13.14 – The Kubernetes page in the Google Cloud web portal

Again, as you might have already guessed, the first step to launching the cluster is to click on the **Create cluster** button. You will then be presented with quite a few options; however, we are going to use the defaults, other than the **Name**, which I am going to change to **mastering-docker-gke**. Once the name has been changed, click on the **Create** button:



Create a Kubernetes cluster

Cluster basics

The new cluster will be created with the name, version and in the location that you specify here. After the cluster is created, name and location can't be changed.

To experiment with an affordable cluster, try **My first cluster** in the **Cluster set-up guides**

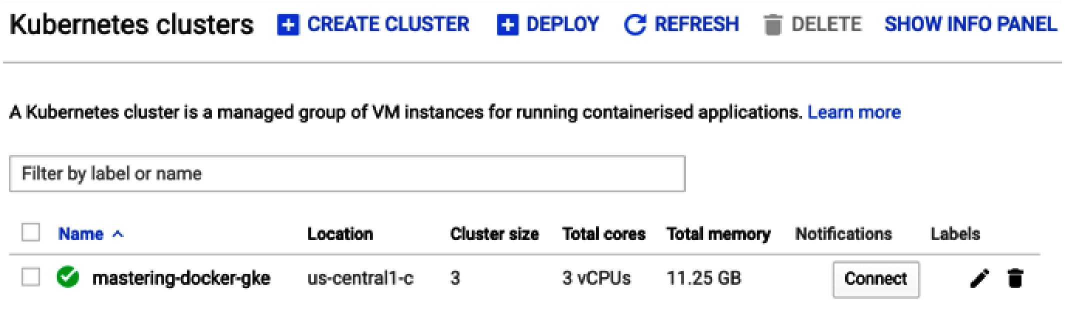
Name: mastering-docker-gke

Location type
● Zonal
○ Regional

Zone: us-central1-c

Figure 13.15 – Entering the cluster details

Launching the cluster will take around 10 minutes. Once it is created, you should see something like the following:



Kubernetes clusters

A Kubernetes cluster is a managed group of VM instances for running containerised applications. Learn more

□	Name ^	Location	Cluster size	Total cores	Total memory	Notifications	Labels
□	✓ mastering-docker-gke	us-central1-c	3	3 vCPUs	11.25 GB		

Figure 13.16 – The cluster is ready

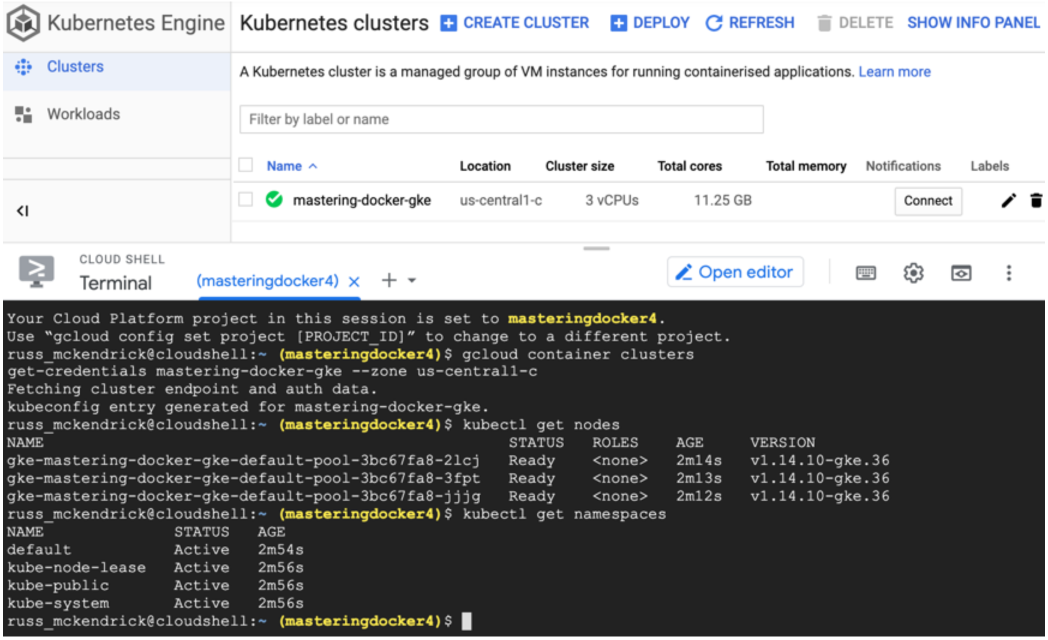
Like Microsoft Azure, Google Cloud has a shell built into the web interface, and the shell has kubectl and the gcloud command-line tool installed and configured. Open the shell by clicking on the >_ button in the top right, then type in the following command to configure how kubectl accesses your cluster:

```
$ gcloud container clusters get-credentials mastering-docker-
gke --zone us-central1-c
```

Update the name and zone if you used anything other than what we set it to. Once configured, you should be able to run both of the following commands:

```
$ kubectl get nodes
$ kubectl get namespaces
```

To test the connectivity, if everything has gone as planned, you should see something like the following:



A Kubernetes cluster is a managed group of VM instances for running containerised applications. Learn more

Name	Location	Cluster size	Total cores	Total memory	Notifications	Labels
mastering-docker-gke	us-central1-c	3 vCPUs	11.25 GB			

```
Your Cloud Platform project in this session is set to masteringdocker4.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
russ_mckendrick@cloudshell:~ (masteringdocker4)$ gcloud container clusters
get-credentials mastering-docker-gke --zone us-central1-c
Fetching cluster endpoint and auth data.
kubeconfig entry generated for mastering-docker-gke.
russ_mckendrick@cloudshell:~ (masteringdocker4)$ kubectl get nodes
NAME                                                  STATUS   ROLES    AGE    VERSION
gke-mastering-docker-gke-default-pool-3bc67fa8-21cj   Ready    <none>   2m14s  v1.14.10-gke.36
gke-mastering-docker-gke-default-pool-3bc67fa8-3fpt   Ready    <none>   2m13s  v1.14.10-gke.36
gke-mastering-docker-gke-default-pool-3bc67fa8-jjjg   Ready    <none>   2m12s  v1.14.10-gke.36
russ_mckendrick@cloudshell:~ (masteringdocker4)$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   2m54s
kube-node-lease   Active   2m56s
kube-public       Active   2m56s
kube-system       Active   2m56s
russ_mckendrick@cloudshell:~ (masteringdocker4)$ 
```

Figure 13.17 – Connecting kubectl to our cluster

Launching a basic cluster using the Google Cloud web portal is a really simple process; however, using the gcloud CLI is just as easy.

Launching a cluster using the command-line tools

The `gcloud` command is provided by the Google Cloud SDK package. To install this on macOS using Homebrew and Cask, run the following command:

```
$ brew cask install google-cloud-sdk
```

If you are running Windows, then you can download the installer from `https://dl.google.com/dl/cloudsdk/channels/rapid/GoogleCloudSDKInstaller.exe`.

Linux users running a Debian-based operating system need to run the following:

```
$ echo 'deb [signed-by=/usr/share/keyrings/cloud.google.gpg]
http://packages.cloud.google.com/apt cloud-sdk main' | sudo tee
 -a /etc/apt/sources.list.d/google-cloud-sdk.list
$ curl https://packages.cloud.google.com/apt/doc/apt-key.gpg |
sudo apt-key --keyring /usr/share/keyrings/cloud.google.gpg add
 -
$ sudo apt-get update && sudo apt-get install google-cloud-sdk
```

Finally, Linux users running Red-Hat-based operating systems need to run the following:

```
$ sudo tee -a /etc/yum.repos.d/google-cloud-sdk.repo << EOM
[google-cloud-sdk]
name=Google Cloud SDK
baseurl=https://packages.cloud.google.com/yum/repos/cloud-sdk-
el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-
key.gpg
EOM
```

Once the repository file is in place, you can install the package using the following command:

```
$ sudo yum install google-cloud-sdk
```

Should you have any problems with any of the preceding commands, a link for the quick starts for each of them containing common troubleshooting tips can be found in the further reading section of this chapter.

Once installed, you need to log in and then let the command-line tool know which project to use. You can do this by running the following commands:

```
$ gcloud auth login
$ gcloud config set project masteringdocker4
```

As you may have guessed, the project I am using is called **masteringdocker4**; you should update this to reflect your own project name.

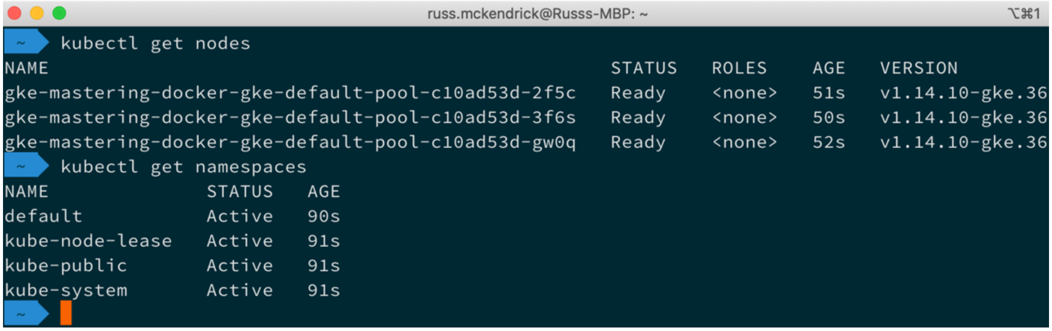
Now that we have `gcloud` configured, we can launch our cluster. To do this, run the following:

```
$ gcloud container clusters create mastering-docker-gke
--num-nodes=3 --zone=us-central1-c
```

This will create a three-node cluster in the US Central 1c zone. Once launched, your kubectl will automatically be configured to communicate with the cluster, meaning that all you need to do is test connectivity by running the following:

```
$ kubectl get nodes
$ kubectl get namespaces
```

This should show you something similar to the following output:



```
~  kubectl get nodes
NAME                                             STATUS   ROLES    AGE   VERSION
gke-mastering-docker-gke-default-pool-c10ad53d-2f5c   Ready    <none>   51s   v1.14.10-gke.36
gke-mastering-docker-gke-default-pool-c10ad53d-3f6s   Ready    <none>   50s   v1.14.10-gke.36
gke-mastering-docker-gke-default-pool-c10ad53d-gw0q   Ready    <none>   52s   v1.14.10-gke.36
~  kubectl get namespaces
NAME              STATUS   AGE
default           Active   90s
kube-node-lease   Active   91s
kube-public       Active   91s
kube-system       Active   91s
~
```

Figure 13.18 – Testing the connection to the Google Kubernetes Engine cluster

Now that we have a Google Kubernetes Engine cluster up and running, we can now launch our application.

Launching an application

As we covered the launching of the application in detail in the *Microsoft Azure Kubernetes Service* section of this chapter, we are not going to go into much detail here, other than to say that you need to run the following commands to launch the app:

```
$ kubectl create namespace sock-shop
$ kubectl -n sock-shop apply  -f 'https://github.com/
microservices-demo/microservices-demo/blob/master/deploy/
kubernetes/complete-demo.yaml?raw=true'
```

You can check the status of the pods and services by running the following:

```
$ kubectl -n sock-shop get pods,services
```

Once everything looks as if it's up and running, you can use the following:

```
$ kubectl -n sock-shop expose deployment front-end
--type=LoadBalancer --name=front-end-lb
```

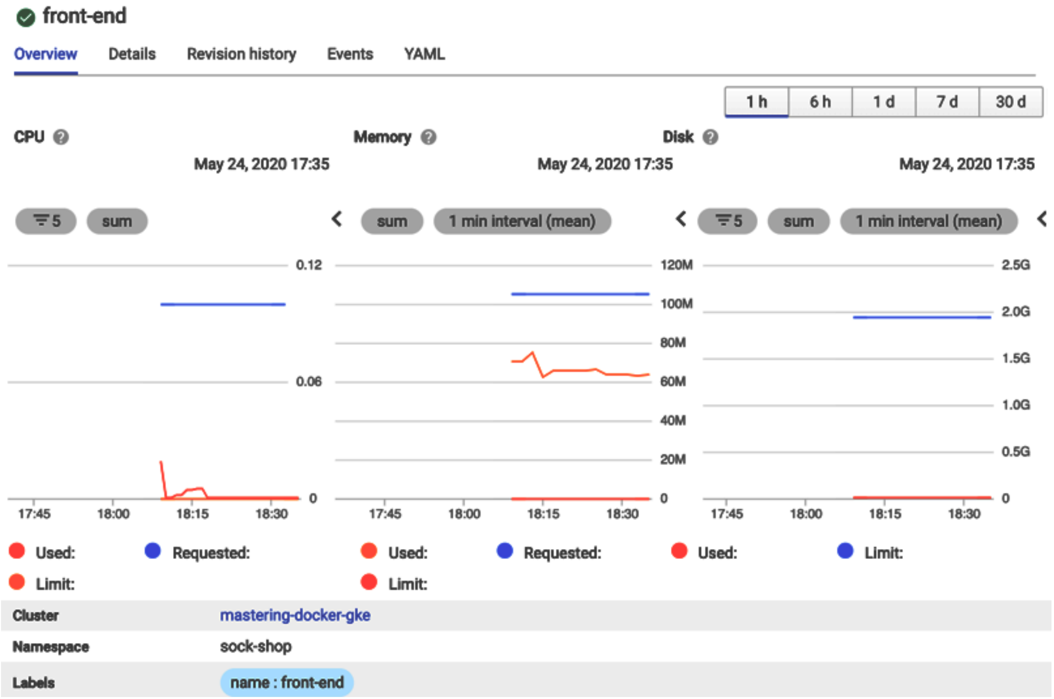
This will expose the application. Once you have entered this, you can enter the following command:

```
$ kubectl -n sock-shop describe services front-end-lb
```

As we mentioned before, to get information on the exposed service you need the `LoadBalancer Ingress` and `Port`. Stick them both in a browser (for example, I used `http://104.154.45.136:8079/`) and you should see that the Sock Shop is up and running.

Cluster information

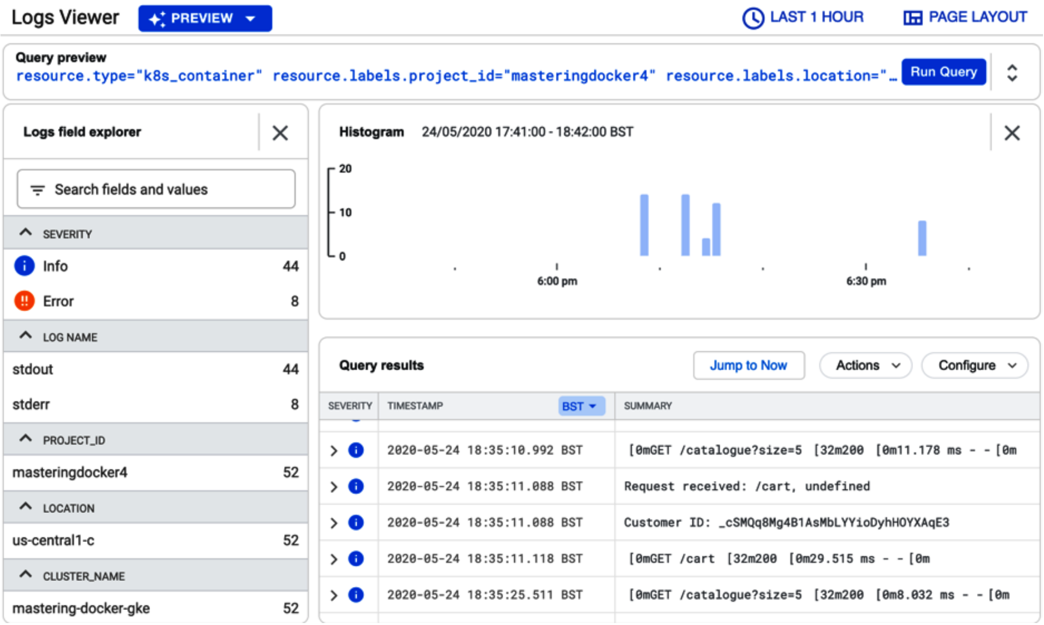
Going back to the Google Cloud web portal and clicking on **Workloads** within your cluster will show you a list of the nonsystem workload. Clicking on the frontend deployment will show you something like the following:



Cluster	mastering-docker-gke
Namespace	sock-shop
Labels	name : front-end

Figure 13.19 – Viewing the deployment details

Clicking on **Container logs**, which can be found at the bottom of the preceding screen, will take you to a page where you can view the logs being generated by the workload:



Query preview

```
resource.type="k8s_container" resource.labels.project_id="masteringdocker4" resource.labels.location="…
```

Histogram 24/05/2020 17:41:00 - 18:42:00 BST

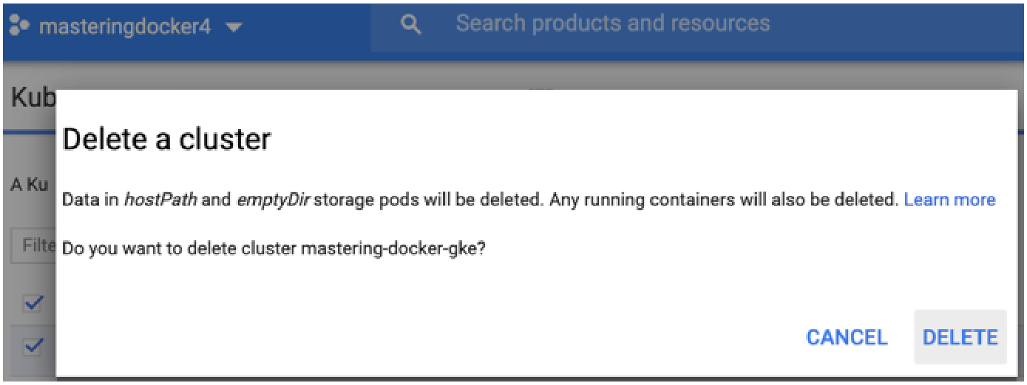
SEVERITY	
Info	44
Error	8
LOG NAME	
stdout	44
stderr	8
PROJECT_ID	
masteringdocker4	52
LOCATION	
us-central1-c	52
CLUSTER_NAME	
mastering-docker-gke	52

Query results

```
SEVERITY  TIMESTAMP                        BST  SUMMARY
          2020-05-24 18:35:10.992 BST           [0mGET /catalogue?size=5 [32m200 [0m11.178 ms - - [0m
          2020-05-24 18:35:11.088 BST           Request received: /cart, undefined
          2020-05-24 18:35:11.088 BST           Customer ID: _cSMQq8Mg4B1AsMbLYYioDyhHOYXAqE3
          2020-05-24 18:35:11.118 BST           [0mGET /cart [32m200 [0m29.515 ms - - [0m
          2020-05-24 18:35:25.511 BST           [0mGET /catalogue?size=5 [32m200 [0m8.032 ms - - [0m
```

Figure 13.20 – Viewing the logs for the frontend deployment

Again, spend a little time having a click around some of the other tabs and options to get an idea of what else you can do with GKE. Once you have finished spending some time getting to know GKE, you should remove all of the resources that were launched. To do this, simply select your cluster from the **Clusters** page of the **Kubernetes Engine** section of the portal and then click on **Delete**. You will be given a warning before the cluster is removed:



Delete a cluster

Data in *hostPath* and *emptyDir* storage pods will be deleted. Any running containers will also be deleted. Learn more

Do you want to delete cluster mastering-docker-gke?

Figure 13.21 – Are you sure you want to delete the cluster?

If you are happy to proceed, then click on **DELETE** and give it about 10 minutes.

Google Kubernetes Engine summary

I am sure that you agree that the Google Kubernetes Service is another simple service to configure, and as you must now be noticing, once the cluster is up and running, interacting with it is a consistent experience. Again, Google offers deep integration with other Google Cloud services, such as its monitoring, databases, and load-balancing services.

Running the cluster at the specifications that we set up would cost around $150 per month; however, $73 of that cost is the GKE cluster management fee, to run the cluster at the same specifications as the Microsoft Azure Kubernetes Service will cost around $220 per month, with the GKE cluster management fee staying the same.

Both Googles and Microsoft Kubernetes offerings have been established for quite a while, so let's take a look at the last of the big three cloud providers' Kubernetes offerings and move onto Amazon Web Services

Amazon Elastic Kubernetes Service (EKS)

The next Kubernetes service we are going to take a look at is the Amazon Elastic Container Service for Kubernetes, or Amazon EKS for short. This is the most recently launched service of the three services we have covered so far. In fact, you could say that Amazon was very late to the Kubernetes party.

Unfortunately, the command-line tools for Amazon are not as user friendly as the ones we used for Microsoft Azure and Google Cloud. Because of this, we are going to be using a tool called `eksctl`, which was written by Weave, the same people who created the demo store we have been using. It has been adopted by Amazon as the official command client of EKS, as opposed to the commands built into their own client.

Because of this, we are going to bypass the web-based portal and concentrate on `eksctl`, which itself makes use of the AWS command-line tools.

Launching a cluster using the command-line tools

Before we install eksctl, we need to install the AWS command-line tools. To do this on macOS using Homebrew, run the following command:

```
$ brew install awscli
```

If you are a Windows user, then you can download the installer from https://awscli.amazonaws.com/AWSCLIV2.msi.

Finally, Linux users can run the following commands to download and install the command-line tools:

```
$ curl 'https://awscli.amazonaws.com/awscli-exe-linux-x86_64.
zip' -o 'awscliv2.zip'
$ unzip awscliv2.zip
$ sudo ./aws/install
```

Once installed, you will need to configure your credentials. Unlike the az and gcloud commands, you need to log in to the AWS web portal to do this. Details on the steps you need to take can be found at https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html.

Once you have the AWS command-line tools installed and configured, you can proceed with installing eksctl, again starting with macOS and Homebrew:

```
$ brew tap weaveworks/tap
$ brew install weaveworks/tap/eksctl
```

It is recommended that Windows users use Chocolatey:

```
$ chocolatey install eksctl
```

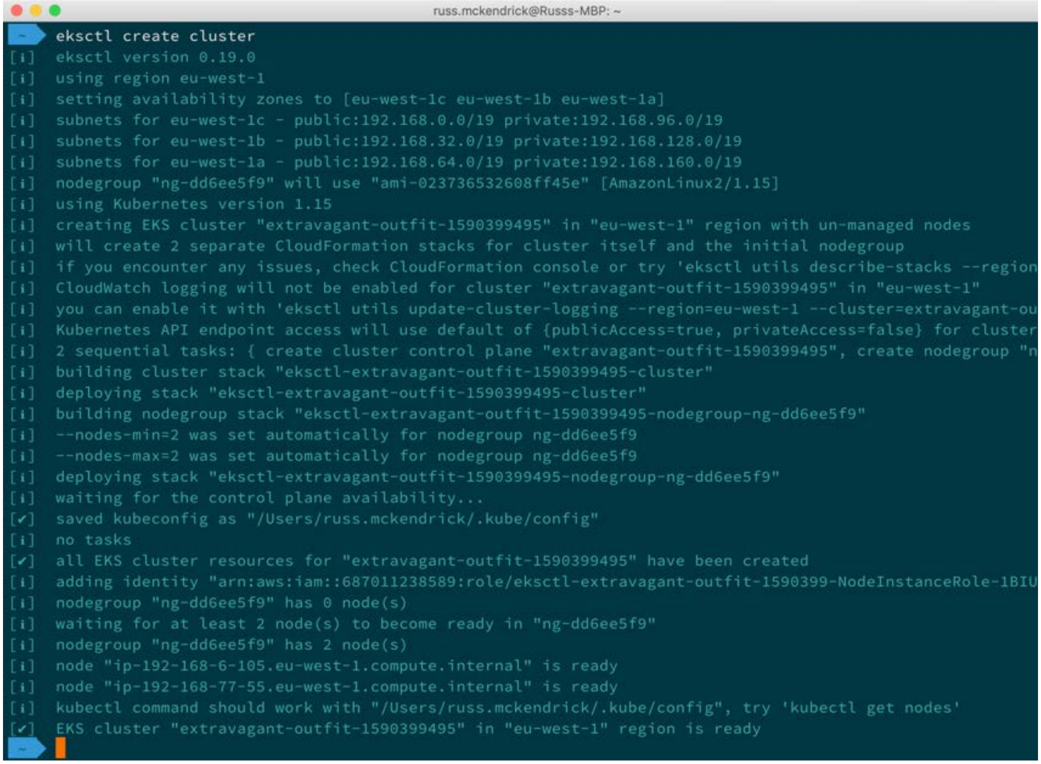
Finally, Linux users can download the precompiled binary straight from GitHub using the following commands:

```
$ curl --silent --location 'https://github.com/weaveworks/
eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.
gz' | tar xz -C /tmp
$ sudo mv /tmp/eksctl /usr/local/bin
```

Once installed, you will be able to run the following command to create and configure your cluster:

```
$ eksctl create cluster
```

Once launched, you should see something like the following, which, as you can see, is quite descriptive:



```
~ eksctl create cluster
[i]  eksctl version 0.19.0
[i]  using region eu-west-1
[i]  setting availability zones to [eu-west-1c eu-west-1b eu-west-1a]
[i]  subnets for eu-west-1c - public:192.168.0.0/19 private:192.168.96.0/19
[i]  subnets for eu-west-1b - public:192.168.32.0/19 private:192.168.128.0/19
[i]  subnets for eu-west-1a - public:192.168.64.0/19 private:192.168.160.0/19
[i]  nodegroup "ng-dd6ee5f9" will use "ami-023736532608ff45e" [AmazonLinux2/1.15]
[i]  using Kubernetes version 1.15
[i]  creating EKS cluster "extravagant-outfit-1590399495" in "eu-west-1" region with un-managed nodes
[i]  will create 2 separate CloudFormation stacks for cluster itself and the initial nodegroup
[i]  if you encounter any issues, check CloudFormation console or try 'eksctl utils describe-stacks --region
[i]  CloudWatch logging will not be enabled for cluster "extravagant-outfit-1590399495" in "eu-west-1"
[i]  you can enable it with 'eksctl utils update-cluster-logging --region=eu-west-1 --cluster=extravagant-ou
[i]  Kubernetes API endpoint access will use default of {publicAccess=true, privateAccess=false} for cluster
[i]  2 sequential tasks: { create cluster control plane "extravagant-outfit-1590399495", create nodegroup "n
[i]  building cluster stack "eksctl-extravagant-outfit-1590399495-cluster"
[i]  deploying stack "eksctl-extravagant-outfit-1590399495-cluster"
[i]  building nodegroup stack "eksctl-extravagant-outfit-1590399495-nodegroup-ng-dd6ee5f9"
[i]  --nodes-min=2 was set automatically for nodegroup ng-dd6ee5f9
[i]  --nodes-max=2 was set automatically for nodegroup ng-dd6ee5f9
[i]  deploying stack "eksctl-extravagant-outfit-1590399495-nodegroup-ng-dd6ee5f9"
[i]  waiting for the control plane availability...
[✔]  saved kubeconfig as "/Users/russ.mckendrick/.kube/config"
[i]  no tasks
[✔]  all EKS cluster resources for "extravagant-outfit-1590399495" have been created
[i]  adding identity "arn:aws:iam::687011238589:role/eksctl-extravagant-outfit-1590399-NodeInstanceRole-1BIU
[i]  nodegroup "ng-dd6ee5f9" has 0 node(s)
[i]  waiting for at least 2 node(s) to become ready in "ng-dd6ee5f9"
[i]  nodegroup "ng-dd6ee5f9" has 2 node(s)
[i]  node "ip-192-168-6-105.eu-west-1.compute.internal" is ready
[i]  node "ip-192-168-77-55.eu-west-1.compute.internal" is ready
[i]  kubectl command should work with "/Users/russ.mckendrick/.kube/config", try 'kubectl get nodes'
[✔]  EKS cluster "extravagant-outfit-1590399495" in "eu-west-1" region is ready
~
```

Figure 13.22 – Launching the EKS cluster using eksctl

As part of the launch, `eksctl` will have configured your local `kubectl` context, meaning that you can run the following:

```
$ kubectl get nodes
$ kubectl get services
```

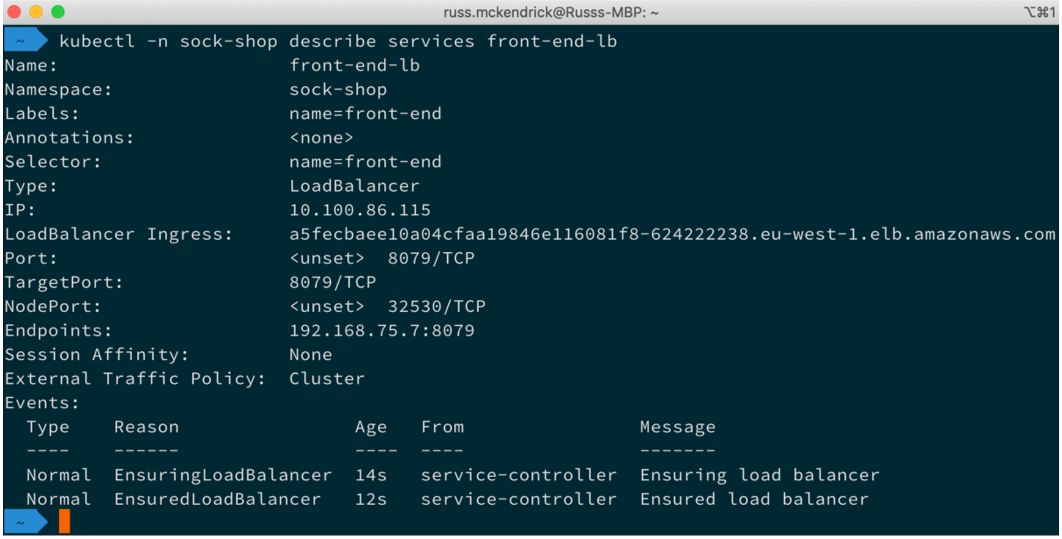
Now that we have the cluster up and running, we can launch the demo store, just like we did previously.

Launching an application

You should be getting quite good at deploying the Sock Shop application now, so here is a recap of all of the commands you need:

```
$ kubectl create namespace sock-shop
$ kubectl -n sock-shop apply -f 'https://github.com/
microservices-demo/microservices-demo/blob/master/deploy/
kubernetes/complete-demo.yaml?raw=true'
$ kubectl -n sock-shop get pods,services
$ kubectl -n sock-shop expose deployment front-end
--type=LoadBalancer --name=front-end-lb
$ kubectl -n sock-shop describe services front-end-lb
```

You may notice that the **LoadBalancer Ingres** is actually a fully qualified domain rather than an IP address:



```
~  kubectl -n sock-shop describe services front-end-lb
Name:                     front-end-lb
Namespace:                sock-shop
Labels:                   name=front-end
Annotations:              <none>
Selector:                 name=front-end
Type:                     LoadBalancer
IP:                       10.100.86.115
LoadBalancer Ingress:     a5fecbaee10a04cfaa19846e116081f8-624222238.eu-west-1.elb.amazonaws.com
Port:                     <unset>  8079/TCP
TargetPort:               8079/TCP
NodePort:                 <unset>  32530/TCP
Endpoints:                192.168.75.7:8079
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason                Age   From                Message
  ----    ------                ----  ----                -------
  Normal  EnsuringLoadBalancer  14s   service-controller  Ensuring load balancer
  Normal  EnsuredLoadBalancer   12s   service-controller  Ensured load balancer
~  
```

Figure 13.23 – Viewing details on the exposed service

You might need to give it a few minutes for the DNS for the load balancer to update, but you should be able to use the URL and port. For my cluster, I went to `http://a5fecbaee10a04cfaa19846e116081f8-624222238.eu-west-1.elb.amazonaws.com:8079` to access the Sock Shop.

Cluster information

If I am honest, this section is pretty redundant as there isn't much information on the cluster exposed in the AWS web portal by default outside of the basic information on the VMs that make up the node cluster. Sure, you can enable some features in CloudWatch to start monitoring your cluster, but this is not done by default at the moment.

The following command will immediately delete the cluster for you:

```
$ eksctl get cluster
$ eksctl delete cluster --name=extravagant-outfit-1590399495
```

However, I recommend double-checking for any resources left over in your AWS web portal, just to make sure you don't get any unexpected bills.

Amazon Elastic Kubernetes Service summary

As already mentioned, the Amazon Elastic Kubernetes Service was the last of the big three public cloud providers' Kubernetes-as-a-service offerings to launch, and in my personal opinion, it is the weakest of the three offerings.

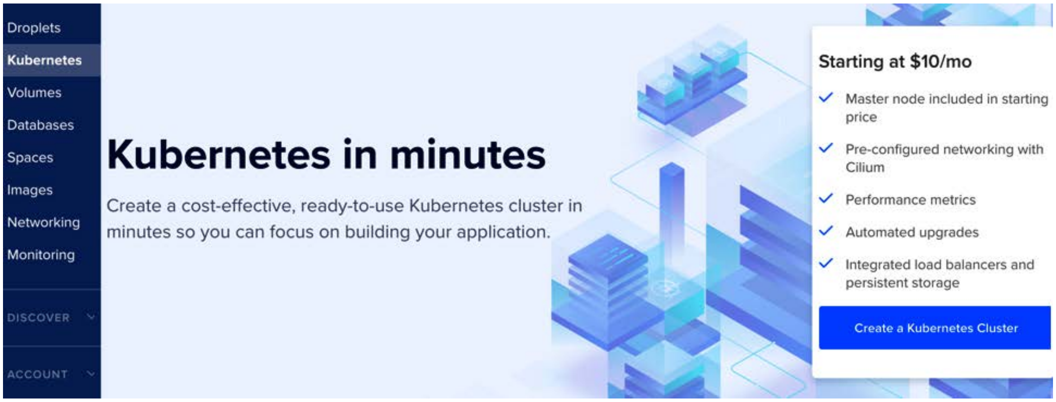
While it is a CNCF-certified hosting platform, it simply doesn't feel as integrated or intuitive as the offerings from Microsoft and Google—everything just feels tacked on to the normally good AWS services; the same can also be said of the Amazon Container Service.

Costwise, the instances are charged at standard EC2 rates, and there is a charge of $0.10 per hour per Kubernetes cluster.

Before we finish the chapter, let's look at one more Kubernetes offering, this time from DigitalOcean.

DigitalOcean Kubernetes

As we are nearing the end of the chapter, we are going to very quickly take a look at the Kubernetes offering from DigitalOcean, as it represents good value for money and is simple to configure. To start with, log in to your DigitalOcean web portal and under the **Manage** section of the right-hand menu, click on **Kubernetes**:



Kubernetes in minutes

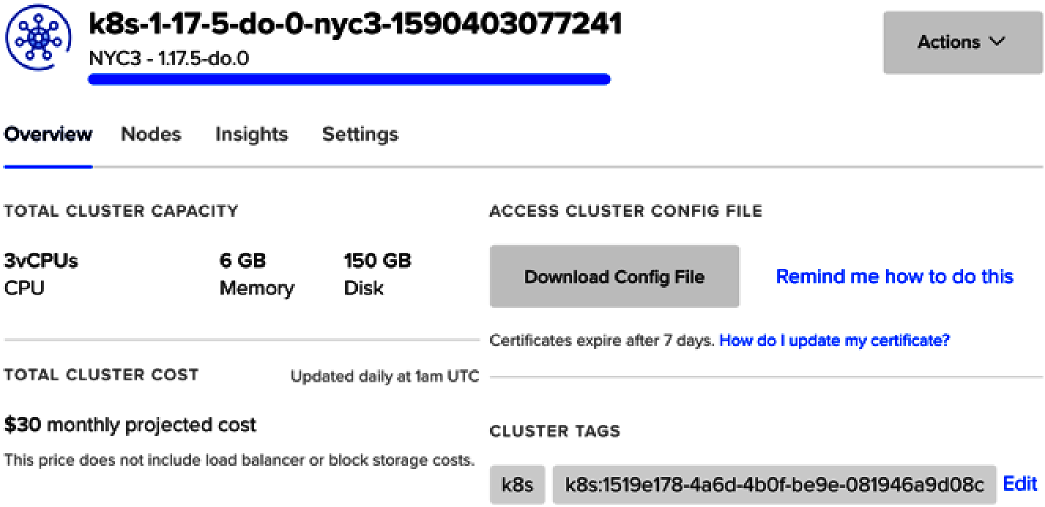
Create a cost-effective, ready-to-use Kubernetes cluster in minutes so you can focus on building your application.

Starting at $10/mo

- ✓ Master node included in starting price
- ✓ Pre-configured networking with Cilium
- ✓ Performance metrics
- ✓ Automated upgrades
- ✓ Integrated load balancers and persistent storage

Figure 13.24 – Details of the DigitalOcean Kubernetes offering

Clicking the **Create a Kubernetes Cluster** button will take you to the **Create a cluster** page, from there just choose a datacenter region and scroll to the bottom of the page; DigitalOcean has some great defaults so we can skim past them. Once at the bottom, click on the **Create Cluster** button and wait. After around five minutes, your cluster will be available:



k8s-1-17-5-do-0-nyc3-1590403077241

NYC3 - 1.17.5-do.0

TOTAL CLUSTER CAPACITY

3vCPUs CPU
6 GB Memory
150 GB Disk

ACCESS CLUSTER CONFIG FILE

Certificates expire after 7 days. How do I update my certificate?

TOTAL CLUSTER COST Updated daily at 1am UTC

$30 monthly projected cost

This price does not include load balancer or block storage costs.

CLUSTER TAGS

k8s k8s:1519e178-4a6d-4b0f-be9e-081946a9d08c

Figure 13.25 – Viewing the newly created DigitalOcean Kubernetes cluster

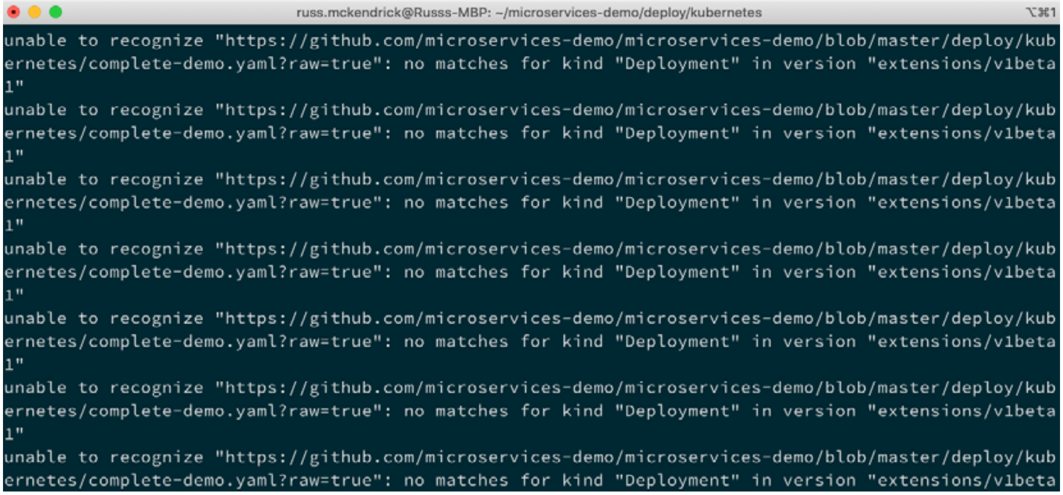
Now that the cluster is up and running, you can click on the **Download Config File** button to grab a copy of the `kubectl` configuration for your cluster, or, if you have it installed (see the further reading section for a link), you can use the `doctl` command to download and configure your local `kubectl` to talk to your newly created cluster.

The command to do this is as follows; please make sure that you update the name at the end to match that of your own cluster:

```
$ doctl kubernetes cluster kubeconfig save k8s-1-17-5-do-0-
nyc3-1590403077241
```

Once configured, you know what to do: launch the Sock Shop application using the same commands that we have been using throughout the chapter—well sort of; if you want to steam ahead and try to launch the application you would receive several errors. Go ahead and try it.

The errors, which look like the following screenshot, are there because of changes to the API in the later version of Kubernetes that DigitalOcean launches by default:


russ.mckendrick@Russs-MBP: ~/microservices-demo/deploy/kubernetes
⌥⌘1

```
unable to recognize "https://github.com/microservices-demo/microservices-demo/blob/master/deploy/kub
ernetes/complete-demo.yaml?raw=true": no matches for kind "Deployment" in version "extensions/v1beta
1"
unable to recognize "https://github.com/microservices-demo/microservices-demo/blob/master/deploy/kub
ernetes/complete-demo.yaml?raw=true": no matches for kind "Deployment" in version "extensions/v1beta
1"
unable to recognize "https://github.com/microservices-demo/microservices-demo/blob/master/deploy/kub
ernetes/complete-demo.yaml?raw=true": no matches for kind "Deployment" in version "extensions/v1beta
1"
unable to recognize "https://github.com/microservices-demo/microservices-demo/blob/master/deploy/kub
ernetes/complete-demo.yaml?raw=true": no matches for kind "Deployment" in version "extensions/v1beta
1"
unable to recognize "https://github.com/microservices-demo/microservices-demo/blob/master/deploy/kub
ernetes/complete-demo.yaml?raw=true": no matches for kind "Deployment" in version "extensions/v1beta
1"
unable to recognize "https://github.com/microservices-demo/microservices-demo/blob/master/deploy/kub
ernetes/complete-demo.yaml?raw=true": no matches for kind "Deployment" in version "extensions/v1beta
1"
unable to recognize "https://github.com/microservices-demo/microservices-demo/blob/master/deploy/kub
ernetes/complete-demo.yaml?raw=true": no matches for kind "Deployment" in version "extensions/v1beta
```

Figure 13.26 – Errors with the Sock Shop

Never fear though—this is easily resolved. First, let's remove the namespace with the partly deployed application in it:

```
$ kubectl delete namespace sock-shop
```

Once these are removed, we need to clone the Sock Shop repository and recreate the namespace. To do this, run the following command:

```
$ git clone https://github.com/microservices-demo/
microservices-demo.git
$ kubectl create namespace sock-shop
```

Next, we have to change directory to the Kubernetes deployment folder, update the definition files, and then launch the application. For this, we need to run the following:

```
$ cd microservices-demo/deploy/kubernetes
$ kubectl convert -f . | kubectl create -f -
```

From there, you can check the pods and services, expose the frontend, and get information on the exposed service using the following:

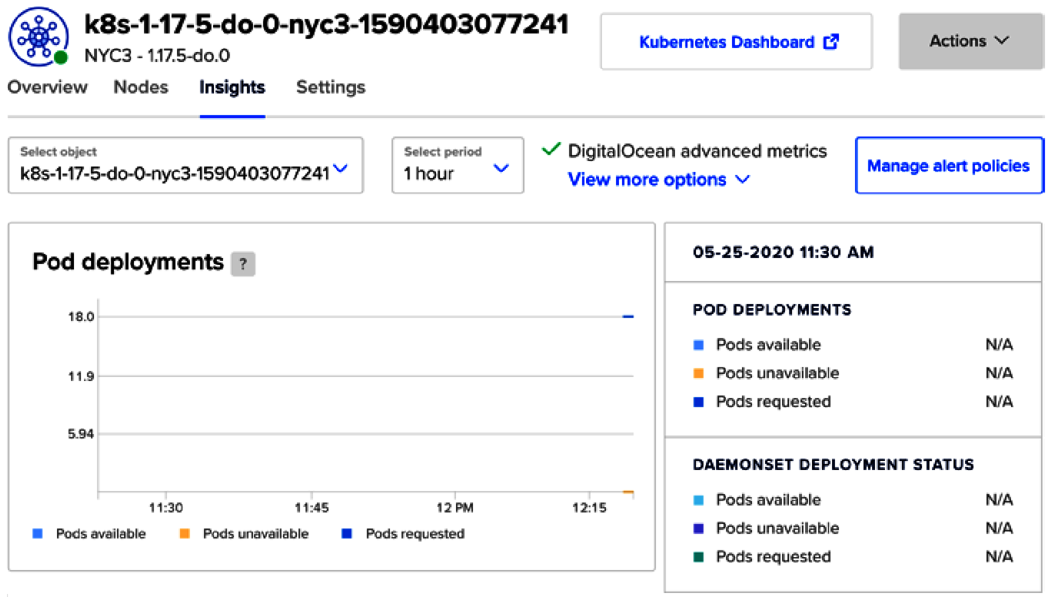
```
$ kubectl -n sock-shop get pods,services
$ kubectl -n sock-shop expose deployment front-end
--type=LoadBalancer --name=front-end-lb
$ kubectl -n sock-shop describe services front-end-lb
```

It may take a minute or two for the services to be exposed as a DigitalOcean load balancer is launched.

Before we return to the DigitalOcean web portal, let's enable the advanced cluster metrics. To do this, we need to deploy another application. You can do this by running the following two commands:

```
$ git clone https://github.com/kubernetes/kube-state-metrics.
git
$ kubectl create -f kube-state-metrics/examples/standard/
```

Once installed, it will take a minute or two for the metrics to appear in the portal. You can find them by selecting your cluster and then clicking on the **Insights** tab:



k8s-1-17-5-do-0-nyc3-1590403077241
NYC3 - 1.17.5-do.0

Select object: k8s-1-17-5-do-0-nyc3-1590403077241
Select period: 1 hour
✓ DigitalOcean advanced metrics

05-25-2020 11:30 AM

POD DEPLOYMENTS

🟦 Pods available	N/A
🟧 Pods unavailable	N/A
🟦 Pods requested	N/A

DAEMONSET DEPLOYMENT STATUS

🟦 Pods available	N/A
🟪 Pods unavailable	N/A
🟩 Pods requested	N/A

Figure 13.27 – Viewing details on the cluster in the DigitalOcean web portal

I would recommend having a click around the DigitalOcean web portal before you remove the cluster, as I am sure you will agree that the service is quite well integrated and comes at a bargain price. The three-node cluster we launched would cost just $30 per month to run!

You can terminate the cluster through the web portal. Also, make sure that you also remove the load balancer as this will remain once the cluster has been terminated to avoid unexpected costs at the end of the month.

Summary

In this chapter, we looked at launching Kubernetes clusters in various clouds and running the same demo application in all of them. I am sure that by the end of the chapter you were quite bored of launching the same application over and over again; however, that was the point.

We have looked at four very different and traditionally incompatible cloud providers and deployed the same application using the same tools and commands across all four of them. Admittedly, we had to make some allowances for the different versions of Kubernetes being used, but for the most part, we didn't have to make any provider-specific allowances once we started working with `kubectl`.

This is one of the key reasons why Kubernetes has become so popular: it truly does allow you to define and distribute your applications in a platform-agnostic way—even just a few years ago, being able to deploy an application locally and then across four public cloud providers using the same set of commands and configurations seemed impossible without a lot of complexity, which I am sure you will agree there has been none of in the last few chapters.

In the next chapter, we are going to go back to Docker and learn how to best secure your containers.

Questions

1. Once our cluster launches, what command did we need to run to create the namespace for the Sock Shop store?
2. How do you find out the full details of the Load Balancer?
3. Name the official Amazon Elastic Kubernetes Service CLI.

Further reading

The product pages for each of the Kubernetes services can be found at the following links:

- **Azure Kubernetes Service:** `https://azure.microsoft.com/en-gb/services/kubernetes-service/`
- **Google Kubernetes Engine:** `https://cloud.google.com/kubernetes-engine/`
- **Amazon Elastic Container Service for Kubernetes:** `https://aws.amazon.com/eks/`
- **DigitalOcean Kubernetes:** `https://www.digitalocean.com/products/kubernetes/`

Quick starts for the various command-line tools used in the chapter can be found at the following links:

- **Azure CLI:** `https://docs.microsoft.com/en-us/cli/azure/?view=azure-cli-latest`
- **Google Cloud SDK:** `https://cloud.google.com/sdk/`
- **AWS Command-Line Interface:** `https://aws.amazon.com/cli/`
- **eksctl, the official CLI for Amazon EKS:** `https://eksctl.io/`
- **doctl:** `https://github.com/digitalocean/doctl/`

Finally, for more details on the demo store, go to the following link:

- **Sock Shop:** `https://microservices-demo.github.io/`

Section 3: Best Practices

In this section, we will take the skills learned from the previous chapters and cover how they can be used in the real world.

This section comprises the following chapters:

Chapter 14, Securing Your Docker Environment

Chapter 15, Docker Workflows

Chapter 16, Next Steps with Docker

14
Docker Security

In this chapter, we will take a look at Docker security, a topic at the forefront of everyone's mind these days. We will split the chapter up into the following five sections:

- Container considerations
- Docker commands
- Best practices
- The Docker Bench security "applications"
- Third-party security services

Technical requirements

In this chapter, we will be using Docker Desktop, and we will be using Multipass to launch a Docker host that we will then launch some poorly configured container on. As in the previous chapters, I will be using my preferred operating system, which is macOS.

As previously, the Docker commands that we will run will work on all three of the operating systems that we have installed Docker on so far. However, some of the supporting commands, which will be few and far between, may only apply to macOS and Linux-based operating systems.

Check out the following video to see the Code in Action:
`https://bit.ly/3m8ubVd`

Container considerations

When Docker was first released, there was a lot of talk about Docker versus virtual machines. I remember reading articles in magazines, commenting on threads on Reddit, and reading endless blog posts. In the early days of the Docker alpha and beta versions, people used to approach Docker containers like virtual machines, because there weren't really any other points of reference, and we viewed them as tiny VMs.

In the past, I would enable SSH, run multiple processes in containers, and even create my container images by launching a container and running the commands to install my software stack. We discussed in *Chapter 2, Building Container Images*, that you should never install, configure, and export SSH on your containers as it is regarded as a bad practice as Docker provides ways for you to access your containers without having to use SSH.

So, rather than discussing containers versus virtual machines, let's look at some of the considerations that you need to make when running containers, rather than virtual machines.

The advantages

When you start a Docker container, Docker Engine does a lot of work behind the scenes. One of the tasks that Docker Engine performs when launching your containers is setting up namespaces and control groups. What does that mean? By setting up namespaces, Docker keeps the processes isolated in each container, not only from other containers but also from the host system. The control groups ensure that each container gets its own share of items, such as CPU, memory, and disk `I/O`. More importantly, they ensure that one container doesn't exhaust all of the resources on a given Docker host.

As we saw in *Chapter 4, Managing Containers*, being able to launch your containers in a Docker controlled network means that you can isolate your containers at the application level; all of the containers for Application A will not have any access, at the network layer, to the containers for Application B.

Additionally, this network isolation can run on a single Docker host by using the default network driver, or it can span multiple Docker hosts by using Docker Swarm's built-in, multi-host networking driver, or the Weave Net driver from Weave.

Lastly, what I consider one of the most significant advantages of Docker over a typical virtual machine is that you shouldn't have to log in to the container. Docker is trying its hardest to keep you from needing to log in to a container to manage the process that it is running. With commands such as `docker container exec`, `docker container top`, `docker container logs`, and `docker container stats`, you can do everything that you need to do, without exposing any more services than you have to.

Your Docker hosts

When you are dealing with virtual machines, you can control who has access to which virtual machine. Let's suppose that you only want **User 1**, who is a developer, to have access to the development VMs.

However, **User 2** is an operator who is responsible for both the development and production environments, so they need access to all of the VMs. Most virtual machine management tools allow you to grant role-based access to your VMs.

With Docker, you are at a slight disadvantage because everyone who has access to Docker Engine on your host, either through being granted sudo access or by having their user added to the Docker Linux group, will have access to every Docker container that you are running on that host.

They can run new containers, they can stop existing containers, and they can delete images as well. Be careful to whom you grant permission to access Docker Engine on your hosts. They mostly hold the keys to the kingdom, concerning all of your containers. Knowing this, it is recommended to use Docker hosts only for Docker; keep other services separate from your Docker hosts.

Image trust

If you are running virtual machines, you will most likely be setting them up yourself, from scratch. It's likely that, due to the size of the download (and also the effort in launching it), you will not download a prebuilt machine image that some random person on the internet created. Typically, if you were to do this, it would be a prebuilt virtual appliance from a trusted software vendor.

So, you will be aware of what is inside the virtual machine and what isn't, as you were responsible for building and maintaining it.

Part of the appeal of Docker is its ease of use; however, this ease of use can make it easy to ignore a quite crucial security consideration – do you know what it is running inside your container?

We have already touched upon image trust in earlier chapters. For example, we spoke about not publishing or downloading images that haven't been defined using Dockerfiles, and not embedding custom code or secrets (and so on) directly into an image that you will be pushing to Docker Hub.

While containers have the protection of namespaces, control groups, and network isolation, we discussed how a poorly judged image download could introduce security concerns and risk into your environment. For example, a legitimate container running an unpatched piece of software can add risk to the availability of your application and data.

Now that we have covered some basic principles, let's take a look at the Docker commands that can be used to help tighten up security, as well as to view information about the images that you might be using.

Docker commands

There are two commands that we will be looking at. The first will be the `docker container run` command so that you can see some of the items that you can use to your advantage with this command. Secondly, we will take a look at the `docker container diff` command, which you can use to view what has been done with the image that you are planning to use.

Let's take a look at how we can use these two commands to help secure our containers.

The Docker Run command

With respect to the `docker run` command, we will mainly focus on the option that allows you to set everything inside the container as read-only, instead of a specified directory or volume. This helps to limit the amount of damage that can be caused by malicious "applications" that could also hijack a vulnerable application by updating its binaries.

Let's take a look at how to launch a read-only container, and then break down what it does, as follows:

```
$ docker container run -d --name mysql --read-only -v /
var/lib/mysql -v /tmp -v /var/run/mysqld -e MYSQL_ROOT_
PASSWORD=password mysql
```

Here, we are running a MySQL container and setting the entire container as read-only, except for the following folders:

- `/var/lib/mysql`
- `/var/run/mysqld`
- `/tmp`

These will be created as three separate volumes, and then mounted as read/write. If you do not add these volumes, then MySQL will not be able to start, as it needs read/write access to be able to create the socket file in `/var/run/mysqld`, some temporary files in `/tmp`, and finally, the databases themselves, in `/var/lib/mysql`.

Any other location inside the container won't allow you to write anything in it. If you tried to run the following, it would fail:

```
$ docker container exec mysql touch /trying_to_write_a_file
```

The preceding command would give you the following message:

```
touch: cannot touch '/trying_to_write_a_file': Read-only file
system
```

This can be extremely helpful if you want to control where the containers can write to (or not write to). Be sure to use this wisely. Test thoroughly, as there can be consequences when "applications" can't write to certain locations.

Similar to the previous command, with `docker container run`, where we set everything to read-only (except for a specified volume), we can do the opposite and set just a single volume (or more, if you use more `-v` switches) to read-only.

The thing to remember about volumes is that when you use a volume and mount it in a container, it will mount as an empty volume over the top of the directory inside the container, unless you use the `--volumes-from` switch or add data to the container in some other way after it has been launched; for example, you could use something like the following command (which will not work):

```
$ docker container run -d -v /local/path/to/html/:/var/www/
html/:ro nginx
```

This will mount `/local/path/to/html/` from the Docker host to `/var/www/html/` and will set it to read-only. This can be useful if you don't want a running container to write to a volume, to keep the data or configuration files intact.

The docker diff command

Let's take another look at the `docker diff` command; since it relates to the security aspects of containers, you may want to use the images that are hosted on Docker Hub or other related repositories.

Remember that whoever has access to your Docker host and the Docker daemon has access to all of your running Docker containers. That being said, if you don't have monitoring in place, someone could be executing commands against your containers and doing malicious things.

Let's take a look at the MySQL container that we launched in the previous section:

```
$ docker container diff mysql
```

You will notice that no files are returned. Why is that?

Well, the `docker diff` command tells you the changes that have been made to the image since the container was launched. In the previous section, we launched the MySQL container with the image read-only and then mounted volumes to where we knew MySQL would need to be able to read and write – meaning that there are no file differences between the image that we downloaded and the container that we are running.

Stop and remove the MySQL container, then prune the volumes by running the following:

```
$ docker container stop mysql
$ docker container rm mysql
$ docker volume prune
```

Then, launch the same container again, minus the read-only flag and volumes; this gives us a different story, as follows:

```
$ docker container run -d --name mysql -e MYSQL_ROOT_
PASSWORD=password mysql
$ docker container exec mysql touch /trying_to_write_a_file
$ docker container diff mysql
```

As you can see, there were two folders created, and several files added:

```
C /run
C /run/mysqld
A /run/mysqld/mysqld.pid
A /run/mysqld/mysqld.sock
A /run/mysqld/mysqld.sock.lock
A /run/mysqld/mysqlx.sock
A /run/mysqld/mysqlx.sock.lock
A /trying_to_write_a_file
```

This is a great way to spot anything untoward or unexpected that may be going on within your container. Now that we have looked at how we can launch our containers more securely, let's discuss some of the other best practices we can apply.

Best practices

In this section, we will look at best practices when it comes to Docker. Some of these we have already mentioned in previous chapters. We will then discuss the **Center for Internet Security** guide, which documents how to properly secure all aspects of your Docker environment.

Docker best practices

Before we dive into the Center for Internet Security guide, let's go over some of the best practices for using Docker, as follows:

- **Only launch one application per container**: Docker was built for this, and it makes everything easier, at the end of the day. The isolation that we discussed earlier is where this is key.
- **Only install what you need**: As we already covered in previous chapters, if you have to install more services to support the one process your container should be running, I would recommend that you review the reasons why. This not only keeps your images small and portable, but it also reduces the potential attack surface.
- **Review who has access to your Docker hosts**: Remember, whoever has root or sudo access to your Docker hosts has access to manipulate all of the images and running containers on the host, as well the ability to launch new ones.
- **Always use the latest version of Docker**: This will ensure that all security holes have been patched and that you have the latest features as well. While fixing security issues, keeping up to date using the community version may introduce problems caused by changes in functionality or new features. If this is a concern for you, then you might want to look at the LTS Enterprise versions available from Docker.
- **Use the resources available if you need help**: The Docker community is huge and immensely helpful. Use their website, documentation, and the Slack chat rooms to your advantage when planning your Docker environment and assessing platforms. For more information on how to access Slack and other parts of the community, see *Chapter 16*, *Next Steps with Docker*.

The Center for Internet Security benchmark

The Center for Internet Security (CIS) is an independent, non-profit organization whose goal is to provide a secure online experience. They publish benchmarks and controls that are considered best practices for all aspects of IT.

The CIS benchmark for Docker is available for download, for free. You should note that it is currently a 257-page PDF, released under the Creative Commons license, and it covers Docker CE 18.09 and later.

You will be referring to this guide when you actually run the scan (in the next section of this chapter) and get results back as to what needs to (or should be) fixed. The guide is broken down into the following sections:

- **Host configuration**: This part of the guide is about the configuration of your Docker hosts. This is the part of the Docker environment where all your containers run. Thus, keeping it secure is of the utmost importance. This is the first line of defense against attackers.
- **Docker daemon configuration**: This part of the guide has the recommendations that secure the running Docker daemon. Everything that you do to the Docker daemon configuration affects each and every container. These are the switches that you can attach to the Docker daemon that we saw previously, and to the items you will see in the next section when we run through the tool.
- **Docker daemon configuration files**: This part of the guide deals with the files and directories that the Docker daemon uses. This ranges from permissions to ownership. Sometimes, these areas may contain information that you don't want others to know about, which could be in a plain text format.
- **Container images/runtime and build files**: This part of the guide contains both the information for securing the container images and the build files. The first part contains images, covering base images, and the build files that were used. As we covered previously, you need to be sure about the images that you are using, not only for your base images, but for any aspect of your Docker experience. This section of the guide covers the items that you should follow while creating your own base images.

- **Container runtime**: This section was previously a part of a later section, but it has been moved into its own section in the CIS guide. The container runtime covers a lot of security-related items. Be careful with the runtime variables that you are using. In some cases, attackers can use them to their advantage, when you think you are using them to your own advantage. Exposing too much in your containers, such as exposing application secrets and database connections as environment variables, can compromise the security of not only your container but the Docker host and the other containers running on that host.
- **Docker security operations**: This part of the guide covers the security areas that involve deployment; the items are more closely tied to Docker best practices. Because of this, it is best to follow these recommendations.

The Docker Bench Security application

In this section, we will cover the Docker Bench Security application that you can install and run. The tool will inspect the following:

- The host configuration
- The Docker daemon configuration files
- Container images and build files
- The container runtime
- The Docker security operations Docker Swarm configuration

Looks familiar? It should, as these are the same items that we reviewed in the previous section, only built into an application that will do a lot of the heavy lifting for you. It will show you what warnings arise within your configurations and will provide information on other configuration items, and even the items that have passed the test.

Now, we will look at how to run the tool, a live example, and what the output of the process means.

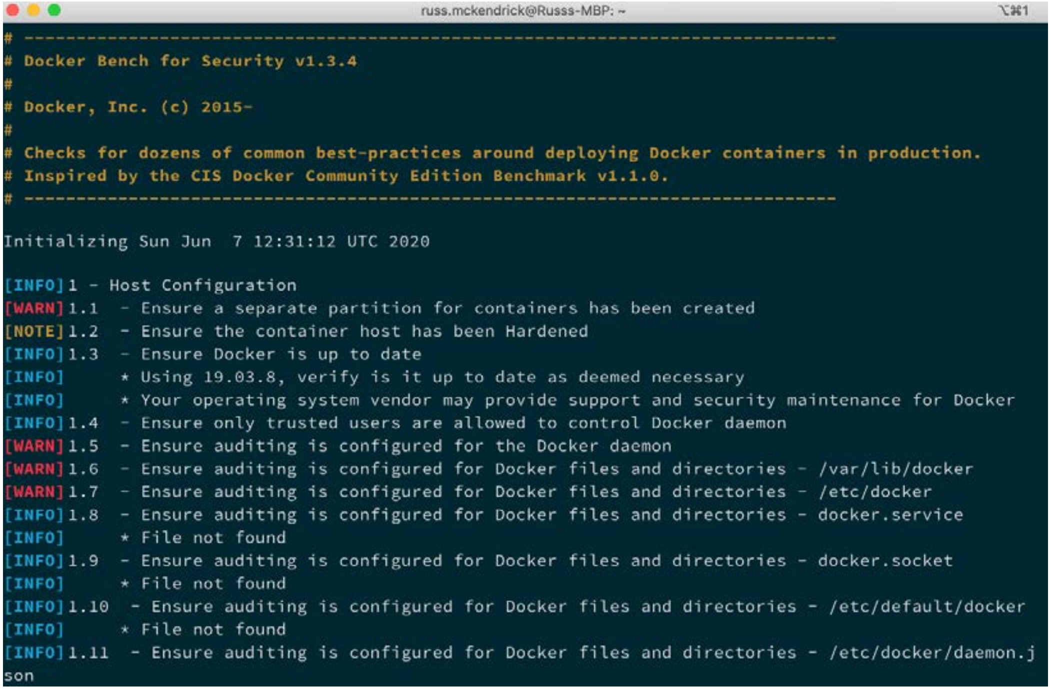
Running the tool on Docker for macOS and Docker for Windows

Running the tool is simple. It's already been packaged for us, inside a Docker container. While you can get the source code and customize the output or manipulate it in some way (say, emailing the output), the default may be all that you need.

The tool's GitHub project can be found at `https://github.com/docker/docker-bench- security/`, and to run the tool on a macOS or Windows machine, you simply have to copy and paste the following into your Terminal. The following command is missing the line needed to check `systemd`, as Moby Linux, which is the underlying operating system for Docker for macOS and Docker for Windows, does not run `systemd`. We will look at a `systemd`-basedsystem when we run the container on an Ubuntu Docker host:

```
docker container run -it --net host --pid host \
    --cap-add audit_control \
    -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
    -v /var/lib:/var/lib \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v /etc:/etc \
    --label docker_bench_security \
    docker/docker-bench-security
```

Once the image has been downloaded, it will launch and immediately start to audit your Docker host, printing the results as it goes, as shown in the following screenshot:



```
# ------------------------------------------------------------------------------
# Docker Bench for Security v1.3.4
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker Community Edition Benchmark v1.1.0.
# ------------------------------------------------------------------------------

Initializing Sun Jun  7 12:31:12 UTC 2020

[INFO] 1 - Host Configuration
[WARN] 1.1  - Ensure a separate partition for containers has been created
[NOTE] 1.2  - Ensure the container host has been Hardened
[INFO] 1.3  - Ensure Docker is up to date
[INFO]      * Using 19.03.8, verify is it up to date as deemed necessary
[INFO]      * Your operating system vendor may provide support and security maintenance for Docker
[INFO] 1.4  - Ensure only trusted users are allowed to control Docker daemon
[WARN] 1.5  - Ensure auditing is configured for the Docker daemon
[WARN] 1.6  - Ensure auditing is configured for Docker files and directories - /var/lib/docker
[WARN] 1.7  - Ensure auditing is configured for Docker files and directories - /etc/docker
[INFO] 1.8  - Ensure auditing is configured for Docker files and directories - docker.service
[INFO]      * File not found
[INFO] 1.9  - Ensure auditing is configured for Docker files and directories - docker.socket
[INFO]      * File not found
[INFO] 1.10  - Ensure auditing is configured for Docker files and directories - /etc/default/docker
[INFO]      * File not found
[INFO] 1.11  - Ensure auditing is configured for Docker files and directories - /etc/docker/daemon.j
son
```

Figure 14.1 – Running a Docker Bench Security check

As you can see, there are a few warnings **[WARN]**, as well as notes **[NOTE]** and information **[INFO]**; however, as this host is managed by Docker, as you would expect, there is not too much to worry about.

Running on Ubuntu Linux

Before we look into the output of the audit in a little more detail, I am going to launch a vanilla Ubuntu server using `multipass` and perform a clean installation of Docker using the official installer by running the following commands:

```
$ multipass launch --name docker-host
$ multipass exec docker-host -- /bin/bash -c 'curl -s https://
get.docker.com | sh - && sudo usermod -aG docker ubuntu'
$ multipass shell docker-host
```

Once installed, I will launch a few containers, all of which don't have very sensible settings. I will launch the following two containers from Docker Hub:

```
$ docker container run -d --name root-nginx -v /:/mnt nginx
$ docker container run -d --name priv-nginx --privileged=true
nginx
```

Then, I will build a custom image, based on Ubuntu 16.04, that runs SSH using the following `Dockerfile`:

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:screencast' | chpasswd
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin
 yes/' /etc/ssh/sshd_config
RUN sed 's@session\s*required\s*pam_loginuid.so@session
optional pam_loginuid.so@g' -i /etc/pam.d/sshd
ENV NOTVISIBLE 'in users profile'
RUN echo 'export VISIBLE=now' >> /etc/profile
EXPOSE 22
CMD ['/usr/sbin/sshd', '-D']
```

I will build and launch the preceding Dockerfile using the following commands:

```
$ docker image build --tag sshd .
$ docker container run -d -P --name sshd sshd
```

As you can see, in one image, we are mounting the root filesystem of our host with full read/write access in the `root-nginx` container. We are also running with extended privileges in `priv-nginx`, and finally, running SSH in `sshd`.

Important note

Please do run the previous Dockerfile or containers outside of this test; we are purposely launching containers that ignore best practices to get results from the scan.

To start the audit on our Ubuntu Docker host, I ran the following:

```
$ docker container run -it --net host --pid host \
    --cap-add audit_control \
    -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
    -v /var/lib:/var/lib \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v /usr/lib/systemd:/usr/lib/systemd \
    -v /etc:/etc --label docker_bench_security \
    docker/docker-bench-security
```

As we are running on an operating system that supports `systemd`, we are mounting `/usr/lib/systemd` so that we can audit it.

There is a lot of output and a lot to digest, but what does it all mean? Let's take a look and break down each section.

Understanding the output

There are four types of output that we will see, as follows:

- `[PASS]`: These items are solid and good to go. They don't need any attention but are good to read, to make you feel warm inside. The more of these, the better!
- `[WARN]`: These are those items that need to be fixed. These are the items that we don't want to see.
- `[INFO]`: These are items that you should review and fix if you feel they are pertinent to your setup and security needs.
- `[NOTE]`: These give best-practice advice.

As mentioned, there are seven main sections that are covered in the audit, as follows:

- Host configuration
- Docker daemon configuration
- Docker daemon configuration files
- Container images and build files
- The container runtime
- Docker security operations
- Docker Swarm configuration

Let's take a look at what we see in each section of the scan. These scan results are from a default Ubuntu Docker host, with no tweaks made to the system at this point. We want to focus on the `[WARN]` items in each section.

Other warnings may come up when you run yours, but these will be the ones that come up for most people (if not for everyone) at first.

Host configuration

I had five items with a `[WARN]` status for my host configuration, as follows:

```
[WARN] 1.1  - Ensure a separate partition for containers has
 been created
```

By default, Docker uses `/var/lib/docker` on the host machine to store all of its files, including all images, containers, and volumes created by the default driver. This means that this folder may grow quickly. As my host machine is running a single partition (and depending on what your containers are doing), this could potentially fill the entire drive, which would render my host machine unusable:

```
[WARN] 1.5  - Ensure auditing is configured for the Docker
daemon
[WARN] 1.6  - Ensure auditing is configured for Docker files
and directories - /var/lib/docker
[WARN] 1.7  - Ensure auditing is configured for Docker files
and directories - /etc/docker
[WARN] 1.10  - Ensure auditing is configured for Docker files
and directories - /etc/default/docker
```

These warnings are being flagged because `auditd` is not installed, and there are no audit rules for the Docker daemon and associated files; for more information on `auditd`, see the blog post at `https://www.linux.com/topic/desktop/customized-file-monitoring-auditd/`.

Docker daemon configuration

My Docker daemon configuration flagged up seven [WARN] statuses, as follows:

```
[WARN] 2.1  - Ensure network traffic is restricted between
containers on the default bridge
```

By default, Docker allows traffic to pass between containers unrestricted, on the same host. It is possible to change this behavior; for more information on Docker networking, see https://docs.docker.com/network/:

```
[WARN] 2.8  - Enable user namespace support
```

By default, the user namespace is not remapped. Mapping them, while possible, can currently cause issues with several Docker features; see https://docs.docker.com/engine/reference/commandline/dockerd/ for more details on known restrictions:

```
[WARN] 2.11  - Ensure that authorization for Docker client
commands is enabled
```

A default installation of Docker allows unrestricted access to the Docker daemon; you can limit access to authenticated users by enabling an authorization plugin. For more details, see https://docs.docker.com/engine/extend/plugins_authorization/:

```
[WARN] 2.12  - Ensure centralized and remote logging is
configured
```

As I am only running a single host, I am not using a service such as rsyslog to ship my Docker host's logs to a central server, nor have I configured a log driver on my Docker daemon; see https://docs.docker.com/config/containers/logging/configure/ for more details:

```
[WARN] 2.14  - Ensure live restore is Enabled
```

The --live-restore flag enables full support of daemon-less containers in Docker; this means that, rather than stopping containers when the daemon shuts down, they continue to run, and it properly reconnects to the containers when restarted.

It is not enabled by default, due to backward compatibility issues. For more details, see https://docs.docker.com/config/containers/live-restore/:

```
[WARN] 2.15  - Ensure Userland Proxy is Disabled
```

There are two ways that your containers can route to the outside world: either by using a hairpin NAT, or a userland proxy. For most installations, the hairpin NAT mode is the preferred mode, as it takes advantage of iptables and has better performance. Where this is not available, Docker uses the userland proxy. Most Docker installations on modern operating systems will support hairpin NAT. For details on how to disable the userland proxy, see https://docs.docker.com/config/containers/container-networking/:

```
[WARN] 2.18  - Ensure containers are restricted from acquiring
new privileges
```

This stops the processes within the containers from potentially gaining any additional privileges by setting `suid` or `sgid` bits; this could limit the impact of any dangerous operations trying to access privileged binaries.

Docker daemon configuration files

I had no `[WARN]` statuses in this section, which is to be expected, as Docker was deployed using the Docker installer.

Container images and build files

I had three `[WARN]` statuses for container images and build files; you may notice that multi-line warnings are prefixed with `*` after the status:

```
[WARN] 4.1  - Ensure a user for the container has been created
[WARN]      * Running as root: sshd
[WARN]      * Running as root: priv-nginx
[WARN]      * Running as root: root-nginx
```

The processes in the containers that I am running are all running as the root user; this is the default action of most containers. For more information, see https://docs.docker.com/engine/security/security/:

```
[WARN] 4.5  - Ensure Content trust for Docker is Enabled
```

Enabling Content Trust for Docker ensures the provenance of the container images that you are pulling, as they are digitally signed when you push them; this means that you are always running the images that you intended to run. For more information on Content Trust, see https://docs.docker.com/engine/security/trust/content_trust/:

```
[WARN] 4.6  - Ensure HEALTHCHECK instructions have been added
to the container image
[WARN]      * No Healthcheck found: [sshd:latest]
[WARN]      * No Healthcheck found: [nginx:latest]
[WARN]      * No Healthcheck found: [ubuntu:16.04]
```

When building your image, it is possible to build in a HEALTHCHECK; this ensures that when a container launches from your image, Docker will periodically check the status of your container and, if needed, it will restart or relaunch it. More details can be found at https://docs.docker.com/engine/reference/builder/#healthcheck.

The container runtime

As we were a little silly when launching our containers on the Docker host that we audited, we know that there will be a lot of vulnerabilities here, and there are 11 of them altogether:

```
[WARN] 5.2  - Ensure SELinux security options are set, if
applicable
[WARN]      * No SecurityOptions Found: sshd
[WARN]      * No SecurityOptions Found: root-nginx
```

The preceding vulnerability is a false positive – we are not running SELinux, as it is an Ubuntu machine, and SELinux is only applicable to Red Hat-based machines. Instead, 5.1 shows us the result, which is a `[PASS]`, which we want:

```
[PASS] 5.1 - Ensure AppArmor Profile is Enabled
```

The next `[WARN]` status is of our own making, as follows:

```
[WARN] 5.4  - Ensure privileged containers are not used
[WARN]      * Container running in Privileged mode: priv-nginx
```

The following is also of our own making:

```
[WARN] 5.6  - Ensure ssh is not run within containers
[WARN]      * Container running sshd: sshd
```

These can be safely ignored; it should be very rare that you have to launch a container running in `Privileged mode`. It is only if your container needs to interact with Docker Engine running on your Docker host; for example, when you are running a GUI (such as Portainer), which we covered in *Chapter 9*, *Portainer – A GUI for Docker*.

We have also discussed that you should not be running SSH in your containers. There are a few use cases, such as running a jump host within a certain network; however, these should be the exception.

The next two `[WARN]` statuses are flagged because, by default on Docker, all running containers on your Docker hosts share the resources equally; setting limits on memory and the CPU priority for your containers will ensure that the containers that you want to have a higher priority are not starved of resources by lower-priority containers:

```
[WARN] 5.10  - Ensure memory usage for container is limited
[WARN]      * Container running without memory restrictions:
sshd
[WARN]      * Container running without memory restrictions:
priv-nginx
[WARN]      * Container running without memory restrictions:
root-nginx
[WARN] 5.11  - Ensure CPU priority is set appropriately on the
container
[WARN]      * Container running without CPU restrictions: sshd
[WARN]      * Container running without CPU restrictions: priv-
nginx
```

```
[WARN]      * Container running without CPU restrictions: root-
nginx
```

As we already discussed earlier in the chapter, if possible, you should be launching your containers as read-only, and mounting volumes for where you know your process needs to write data to:

```
[WARN] 5.12  - Ensure the container's root filesystem is
mounted as read only
[WARN]      * Container running with root FS mounted R/W: sshd
[WARN]      * Container running with root FS mounted R/W: priv-
nginx
[WARN]      * Container running with root FS mounted R/W: root-
nginx
```

The reason the following flags are raised is that we are not telling Docker to bind our exposed port to a specific IP address on the Docker host:

```
[WARN] 5.13  - Ensure incoming container traffic is binded to a
specific host interface
[WARN]      * Port being bound to wildcard IP: 0.0.0.0 in sshd
```

As my test Docker host only has a single NIC, this isn't too much of a problem. However, if my Docker host had multiple interfaces, then this container would be exposed to all of the networks, which could be a problem if I had, for example, an external and internal network. See `https://docs.docker.com/network/` for more details:

```
[WARN] 5.14  - Ensure 'on-failure' container restart policy is
set to '5'
[WARN]      * MaximumRetryCount is not set to 5: sshd
[WARN]      * MaximumRetryCount is not set to 5: priv-nginx
[WARN]      * MaximumRetryCount is not set to 5: root-nginx
```

Although I haven't launched my containers using the `--restart` flag, there is no default value for `MaximumRetryCount`. This means that if a container failed over and over, it would quite happily sit there attempting to restart. This could have a negative effect on the Docker host; adding a `MaximumRetryCount` of `5` will mean that the container will attempt to restart five times before giving up:

```
[WARN] 5.25  - Ensure the container is restricted from
acquiring additional privileges
```

```
[WARN]      * Privileges not restricted: sshd
[WARN]      * Privileges not restricted: priv-nginx
[WARN]      * Privileges not restricted: root-nginx
```

By default, Docker does not put a restriction on a process or its child processes gaining new privileges via `suid` or `sgid` bits. To find out details on how you can stop this behavior, see `https://www.projectatomic.io/blog/2016/03/no-new-privs-docker/`:

```
[WARN] 5.26  - Ensure container health is checked at runtime
[WARN]      * Health check not set: sshd
[WARN]      * Health check not set: priv-nginx
[WARN]      * Health check not set: root-nginx
```

Again, we are not using any health checks, meaning that Docker will not periodically check the status of your containers. To see the GitHub issue for the pull request that introduced this feature, see `https://github.com/moby/moby/pull/22719`:

```
[WARN] 5.28  - Ensure PIDs cgroup limit is used
[WARN]      * PIDs limit not set: sshd
[WARN]      * PIDs limit not set: priv-nginx
[WARN]      * PIDs limit not set: root-nginx
```

Potentially, an attacker could trigger a fork bomb with a single command inside your container. This has the potential to crash your Docker host, and the only way to recover would be to reboot the host. You can protect against this by using the `--pids-limit` flag. For more information, see the pull request at `https://github.com/moby/moby/pull/18697`.

Docker security operations

This section includes `[INFO]` about best practices, as follows:

```
[INFO] 6.1  - Avoid image sprawl
[INFO]      * There are currently: 4 images
[INFO] 6.2  - Avoid container sprawl
[INFO]      * There are currently a total of 4 containers, with
4 of them currently running
```

Docker Swarm configuration

This section includes `[PASS]` information, as we don't have Docker Swarm enabled on the host.

Removing the Multipass machine

Once you have finished with the Ubuntu server, you can remove it by running the following:

```
$ multipass delete docker-host –purge
```

Remember, there is no warning when running the preceding command; it will delete the running machine straight away.

Summing up Docker Bench

As you have seen, running Docker Bench against your Docker host is a much better way to get an understanding of how your Docker host stacks up against the CIS Docker Benchmark; it is certainly a lot more manageable than manually working through every single test in the 257-page document.

Now that we have covered how you can assess and secure your Docker host, let's quickly discuss how we can secure images.

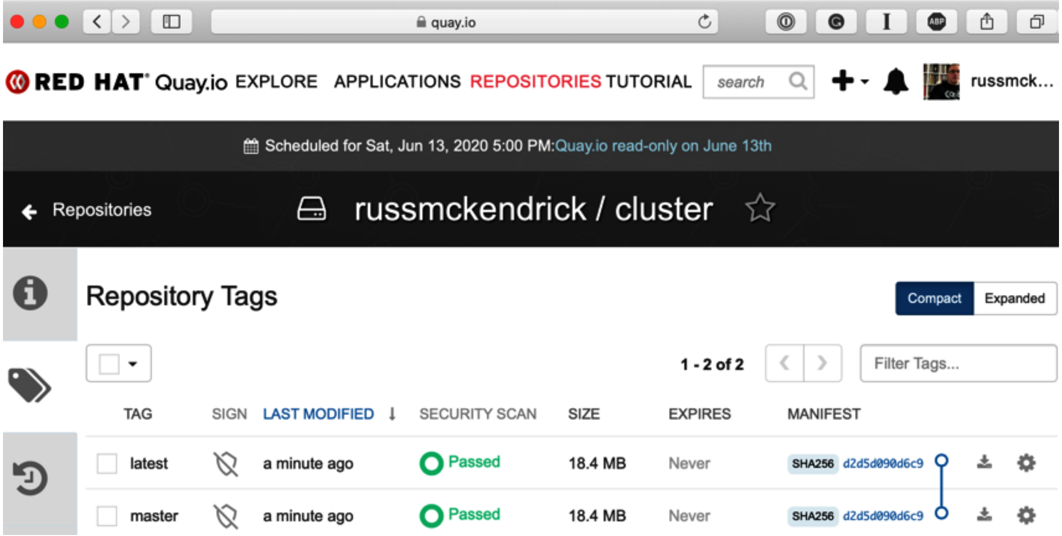
Third-party security services

Before we finish this chapter, we are going to take a look at some of the third-party services available to help you with the vulnerability assessment of your images.

Quay

Quay, an image registry by Red Hat, is similar to Docker Hub/Registry; one difference is that Quay actually performs a security scan of each image after it is pushed/built.

You can see the results of the scan by viewing the **Repository Tags** for your chosen image. Here you will see a column for **Security Scan**. As you can see in the following screenshot, in the example image that we created, there are no problems:



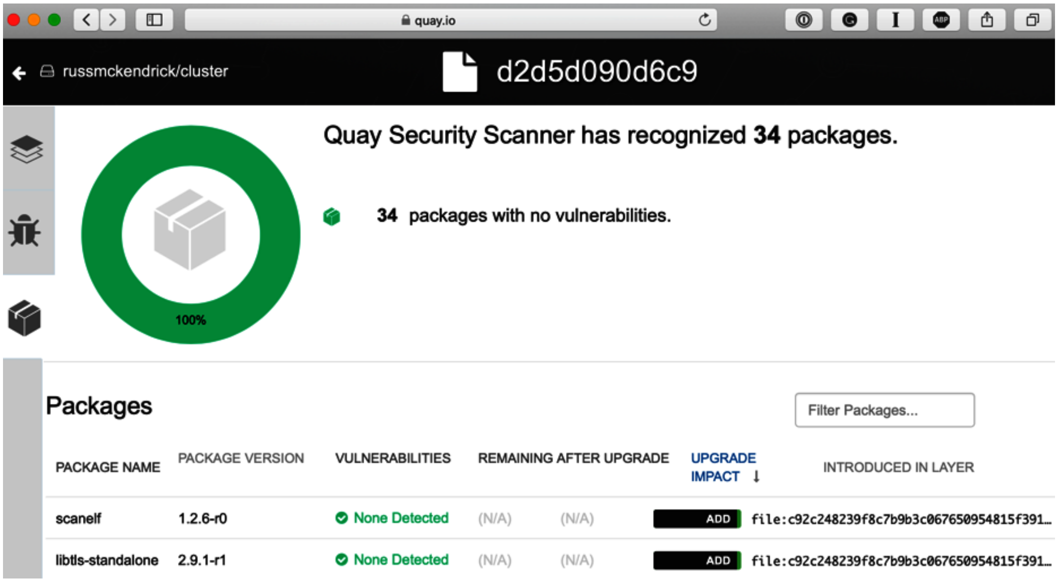
quay.io
RED HAT Quay.io EXPLORE APPLICATIONS REPOSITORIES TUTORIAL
search
russmck...
Scheduled for Sat, Jun 13, 2020 5:00 PM:Quay.io read-only on June 13th
Repositories
russmckendrick / cluster
Repository Tags
Compact
Expanded
1 - 2 of 2
Filter Tags...

TAG	SIGN	LAST MODIFIED	SECURITY SCAN	SIZE	EXPIRES	MANIFEST
□ latest		a minute ago	○ Passed	18.4 MB	Never	SHA256 d2d5d090d6c9
□ master		a minute ago	○ Passed	18.4 MB	Never	SHA256 d2d5d090d6c9

Figure 14.2 – A passed security scan on Quay

Clicking on **Passed** will take you to a more detailed breakdown of any vulnerabilities that have been detected within the image. As there are no vulnerabilities at the moment (which is a good thing), this screen does not tell us much. However, clicking on the **Packages** icon in the left-hand menu will present us with a list of the packages that the scan has discovered.

For our test image, it has found 34 packages with no vulnerabilities, all of which are displayed here, along with confirmation of the version of the package, and how they were introduced to the image:



Quay Security Scanner has recognized **34** packages.

34 packages with no vulnerabilities.

Packages

PACKAGE NAME	PACKAGE VERSION	VULNERABILITIES	REMAINING AFTER UPGRADE	UPGRADE IMPACT ↓	INTRODUCED IN LAYER
scanelf	1.2.6-r0	✓ None Detected	(N/A)	(N/A)	ADD file:c92c248239f8c7b9b3c067650954815f391…
libtls-standalone	2.9.1-r1	✓ None Detected	(N/A)	(N/A)	ADD file:c92c248239f8c7b9b3c067650954815f391…

Figure 14.3 – A list of all packages installed

As you can also see, Quay is scanning our publicly available image, which is being hosted on the free-of-charge open source plan that Quay offers. Security scanning comes as standard with all plans on Quay.

Clair

Clair is an open source project from Red Hat. In essence, it is a service that provides the static analysis functionality for both the hosted version of Quay and the commercially supported, enterprise version.

It works by creating a local mirror of the following vulnerability databases:

- Debian Security Bug Tracker: `https://security-tracker.debian.org/tracker/`
- Ubuntu CVE Tracker: `https://launchpad.net/ubuntu-cve-tracker/`
- Red Hat Security Data: `https://www.redhat.com/security/data/metrics/`
- Oracle Linux Security Data: `https://linux.oracle.com/security/`
- Alpine SecDB: `https://git.alpinelinux.org/cgit/alpine-secdb/`
- NIST NVD: `https://nvd.nist.gov/`

Once it has mirrored the data sources, it mounts the image's filesystem, and then performs a scan of the installed packages, comparing them to the signatures in the preceding data sources.

Clair is not a straightforward service; it only has an API-driven interface, and there are no fancy web-based or command-line tools that ship with Clair by default. The documentation for the API can be found at `https://app.swaggerhub.com/apis/coreos/clair/3.0`.

The installation instructions can be found at the project's GitHub page, at `https://github.com/quay/clair/`.

Also, you can find a list of tools that support Clair on its integration page, at `https://github.com/quay/clair/blob/master/Documentation/integrations.md`.

Before we finish, there is one more tool to look at, and this is one we can run locally.

Anchore

The final tool that we are going to cover is Anchore. This comes in several versions; there are cloud-based offerings and an "on-premises" enterprise version, both of which come with a full, web-based graphical interface. There is a version that hooks into Jenkins, and also the open source command-line scanner, which is what we are going to take a look at now.

This version is distributed as a Docker Compose file, so we will start by creating the folders that we need, and we will also download the Docker Compose file:

```
$ mkdir anchore
$ cd anchore
$ curl https://docs.anchore.com/current/docs/engine/quickstart/
docker-compose.yaml -o docker-compose.yaml
```

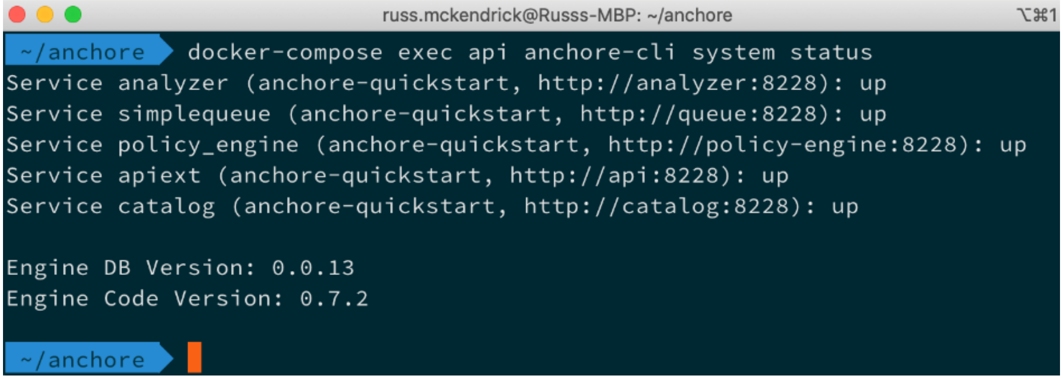
Now that we have the basics in place, you can pull the images and start the containers, as follows:

```
$ docker-compose pull
$ docker-compose up -d
```

Before we can interact with our Anchore deployment, we need the command-line client. Luckily, the Docker Compose file we downloaded comes with a container running the client configured out of the box:

```
$ docker-compose exec api anchore-cli system status
```

This will show you the overall status of your installation; it might take a minute or two from when you first launched for everything to show as up and running:



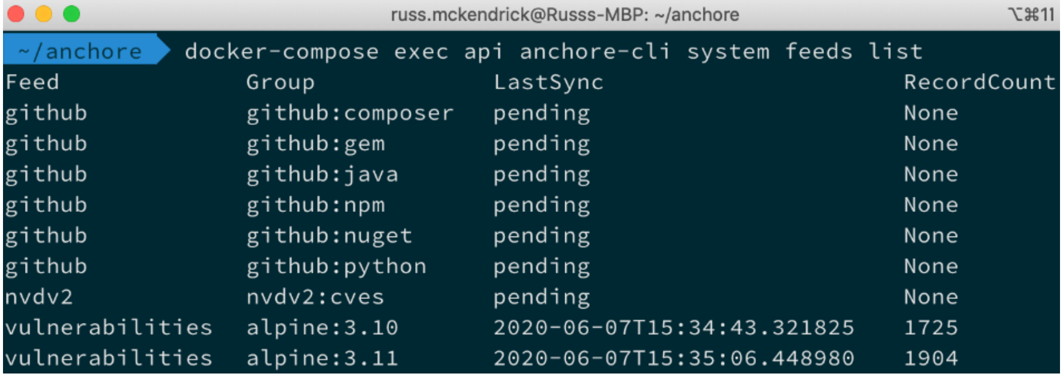
```
~/anchore  docker-compose exec api anchore-cli system status
Service analyzer (anchore-quickstart, http://analyzer:8228): up
Service simplequeue (anchore-quickstart, http://queue:8228): up
Service policy_engine (anchore-quickstart, http://policy-engine:8228): up
Service apiext (anchore-quickstart, http://api:8228): up
Service catalog (anchore-quickstart, http://catalog:8228): up

Engine DB Version: 0.0.13
Engine Code Version: 0.7.2

~/anchore
```

Figure 14.4 – Checking the status of the Anchore engine

The next command shows you where Anchore is in the database sync:

```
$ docker-compose exec api anchore-cli system feeds list
```

As you can see in the following screenshot, my installation is currently syncing the databases. This process can take up to a few hours; however, for our example, we are going to be scanning an Alpine Linux-based database, which are the first databases to be downloaded:



```
~/anchore  docker-compose exec api anchore-cli system feeds list
Feed                 Group                LastSync                       RecordCount
github               github:composer      pending                        None
github               github:gem           pending                        None
github               github:java          pending                        None
github               github:npm           pending                        None
github               github:nuget         pending                        None
github               github:python        pending                        None
nvdv2                nvdv2:cves           pending                        None
vulnerabilities      alpine:3.10          2020-06-07T15:34:43.321825     1725
vulnerabilities      alpine:3.11          2020-06-07T15:35:06.448980     1904
```

Figure 14.5 – Checking the status of the feed download

Next up, we have to grab an image to scan; let's grab an older image, as follows:

```
$ docker-compose exec api anchore-cli image add russmckendrick/
moby-counter:old
```

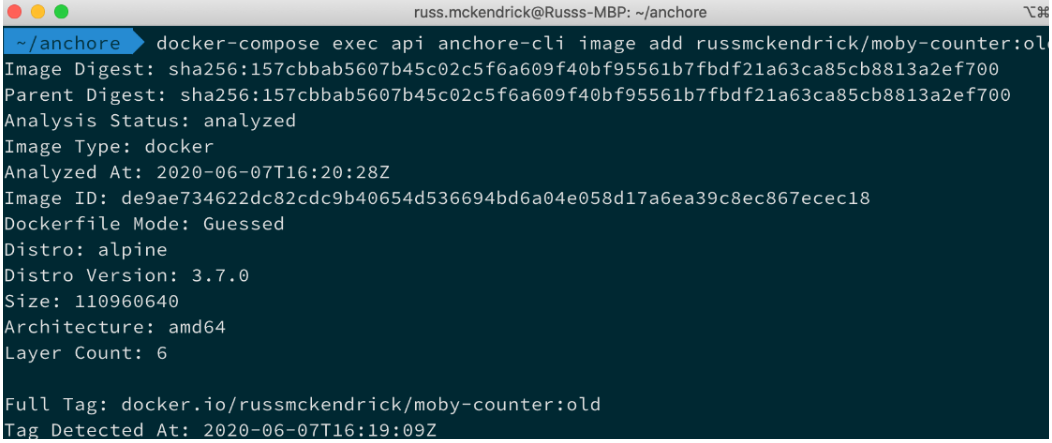
It will take a minute or two to run its initial scan; you can check the status by running the following command:

```
$ docker-compose exec api anchore-cli image list
```

After a while, the status should change from **analyzing** to **analyzed**:

```
$ docker-compose exec api anchore-cli image add russmckendrick/
moby-counter:old
```

This will show you an overview of the image, as follows:



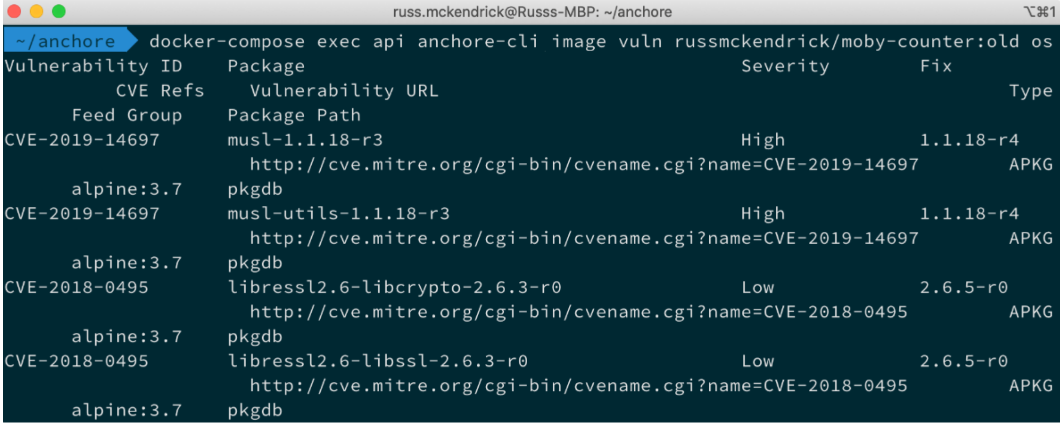
```
~/anchore  docker-compose exec api anchore-cli image add russmckendrick/moby-counter:ol
Image Digest: sha256:157cbbab5607b45c02c5f6a609f40bf95561b7fbdf21a63ca85cb8813a2ef700
Parent Digest: sha256:157cbbab5607b45c02c5f6a609f40bf95561b7fbdf21a63ca85cb8813a2ef700
Analysis Status: analyzed
Image Type: docker
Analyzed At: 2020-06-07T16:20:28Z
Image ID: de9ae734622dc82cdc9b40654d536694bd6a04e058d17a6ea39c8ec867ecec18
Dockerfile Mode: Guessed
Distro: alpine
Distro Version: 3.7.0
Size: 110960640
Architecture: amd64
Layer Count: 6

Full Tag: docker.io/russmckendrick/moby-counter:old
Tag Detected At: 2020-06-07T16:19:09Z
```

Figure 14.6 – Viewing information on the analyzed image

You can then view a list of problems (if there are any) by running the following command:

```
$ docker-compose exec api anchore-cli image vuln
russmckendrick/moby-counter:old os
```

As you can see in the following screenshot, each package that is listed has the current version, a link to the CVE issue, and also confirmation of the version number that fixes the reported issue:



```
~/anchore  docker-compose exec api anchore-cli image vuln russmckendrick/moby-counter:old os
Vulnerability ID    Package                                       Severity        Fix
          CVE Refs    Vulnerability URL                                                   Type
      Feed Group    Package Path
CVE-2019-14697      musl-1.1.18-r3                                High            1.1.18-r4
                      http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14697        APKG
      alpine:3.7    pkgdb
CVE-2019-14697      musl-utils-1.1.18-r3                          High            1.1.18-r4
                      http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14697        APKG
      alpine:3.7    pkgdb
CVE-2018-0495       libressl2.6-libcrypto-2.6.3-r0                Low             2.6.5-r0
                      http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0495         APKG
      alpine:3.7    pkgdb
CVE-2018-0495       libressl2.6-libssl-2.6.3-r0                   Low             2.6.5-r0
                      http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0495         APKG
      alpine:3.7    pkgdb
```

Figure 14.7 – Reviewing the vulnerabilities found by Anchore

You can use the following commands to remove the Anchore containers and database volume:

```
$ docker-compose stop
$ docker-compose rm
$ docker volume rm anchore_anchore-db-volume
```

Also, don't forget to remove the `anchore` folder we created at the start of the section.

Summary

In this chapter, we covered some aspects of Docker security. First, we took a look at some of the things that you must consider when running containers (versus typical virtual machines) with regard to security. We looked at the advantages and your Docker host, and then we discussed image trust. We then took a look at the Docker commands that we can use for security purposes.

We launched a read-only container so that we could minimize any potential damage an intruder could do within our running containers. As not all "applications" lend themselves to running in read-only containers, we then looked at how we can track changes that have been made to the image since it was launched. It is always useful to be able to easily discover any changes that were made on the filesystem at runtime when trying to look into any problems.

Next, we discussed the Center for Internet Security guidelines for Docker. This guide will assist you in setting up multiple aspects of your Docker environment. Lastly, we took a look at Docker Bench Security. We looked at how to get it up and running, and we ran through an example of what the output would look like. We then analyzed the output to see what it meant. Remember the seven items that the application covered: the host configuration, the Docker daemon configuration, the Docker daemon configuration files, the container images and build files, the container runtime, the Docker security operations, and the Docker Swarm configuration.

In the next chapter, we will look at how Docker can fit into your existing workflows, as well as some new ways to approach working with containers.

Questions

1. When launching a container, how can we make all of it, or parts of it, read-only?
2. How many processes should you be running per container?
3. What is the best way to check your Docker installation against the CIS Docker benchmark?
4. When running the Docker Bench Security application, what should be mounted?
5. True or false: Quay only supports image scanning for private images

Further reading

For more information, visit the website at `https://www.cisecurity.org/`. The Docker Benchmark can be found at `https://www.cisecurity.org/benchmark/docker/`.

15
Docker Workflows

By now, you should already be thinking about how you can start to introduce Docker into your everyday workflow. In this chapter, we'll put all the pieces together so you can start using Docker in your local development environment. We'll also look at some of the considerations that you need to take when planning your production environments.

We will be covering the following topics in this chapter, all of which will build on what we have learned in the previous chapters:

- Docker for development
- Monitoring Docker and Kubernetes
- What does production look like?

Technical requirements

In this chapter, we will be using Docker on the desktop. As with previous chapters, I will be using my preferred operating system, which is macOS. The Docker commands we will be running will work on all three of the operating systems we have installed Docker on so far; however, some of the supporting commands, which will be few and far between, may only apply to macOS- and Linux- based operating systems.

Docker for development

We are going to start our look at the workflows by discussing how Docker can be used to aid developers. Right back at the start of *Chapter 1*, *Docker Overview*, one of the first things we discussed in the *Understanding Docker* section was developers and the, works on my machine, problem. So far, we have not really fully addressed this, so let's do that now.

In this section, we are going to look at how a developer could develop their WordPress project on their local machine using Docker for macOS or Docker for Windows, along with Docker Compose.

The aim of this is for us to launch a WordPress installation, which is what you will do by going through the following steps:

1. Download and install WordPress.
2. Allow access to the WordPress files from desktop editors—such as Atom, Visual Studio Code, or Sublime Text—on your local machine.
3. Configure and manage WordPress using the **WordPress command-line** tool (**WPCLI**). This allows you to stop, start, and even remove containers without losing your work.

Before we launch our WordPress installation, let's take a look at the Docker Compose file, which you can find in the `chapter14/docker-wordpress` folder of the accompanying repository:

```
version: '3'
services:
```

We will be launching four different services, starting with `web`:

```
  web:
    image: nginx:alpine
    ports:
      - '8080:80'
    volumes:
      - './wordpress/web:/var/www/html'
      - './wordpress/nginx.conf:/etc/nginx/conf.d/default.conf'
    depends_on:
      - wordpress
```

Followed by the wordpress service:

```
  wordpress:
    image: wordpress:php7.2-fpm-alpine
    volumes:
      - './wordpress/web:/var/www/html'
    depends_on:
      - mysql
```

Next up, we have the mysql database service:

```
  mysql:
      image: mysql:5
      environment:
        MYSQL_ROOT_PASSWORD: 'wordpress'
        MYSQL_USER: 'wordpress'
        MYSQL_PASSWORD: 'wordpress'
        MYSQL_DATABASE: 'wordpress'
      volumes:
        - './wordpress/mysql:/var/lib/mysql'
```

Finally, we have a supporting service simply called wp:

```
  wp:
    image: wordpress:cli-2-php7.2
    volumes:
      - './wordpress/web:/var/www/html'
      - './wordpress/export:/export'
```

We can visualize the Docker Compose file using the docker-compose-viz tool from PMSIpilot.

To do this, run the following command in the same folder as the docker-compose.yml file:

```
$ docker container run --rm -it --name dcv -v $(pwd):/input
pmsipilot/docker-compose-viz render -m image docker-compose.yml

```

This will output a file called `docker-compose.png`, and you should get something that looks like this:

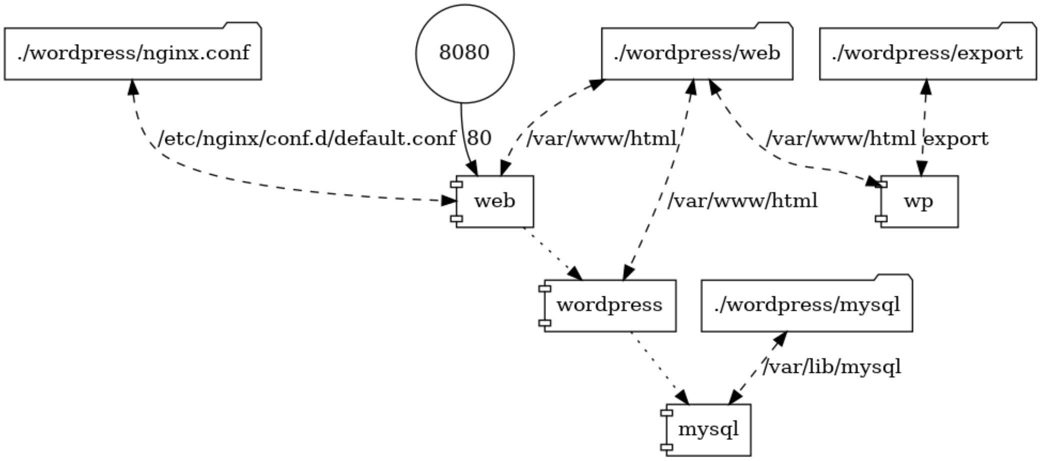


Figure 15.1: Output of the running docker-compose-viz against our WordPress Docker Compose file

The first service is called web. This service is the only one of the four services that is exposed to the host network, and it acts as a frontend to our WordPress installation. It runs the official NGINX image from `https://store.docker.com/images/nginx/` and it performs two roles. Take a look at the NGINX configuration and see if you can guess what they are:

```
server {
  server_name _;
  listen 80 default_server;

  root /var/www/html;
  index index.php index.html;

  access_log /dev/stdout;
  error_log /dev/stdout info;

  location / {
    try_files $uri $uri/ /index.php?$args;
  }

  location ~ .php$ {
    include fastcgi_params;
    fastcgi_pass wordpress:9000;
    fastcgi_index index.php;
    fastcgi_param SCRIPT_FILENAME    $document_root$fastcgi_
script_name;
    fastcgi_buffers 16 16k;
    fastcgi_buffer_size 32k;
  }
}
```

You can see that we are serving all content, apart from PHP, using the NGINX from /var/www/html/, which we are mounting from our host machine using NGINX, and all requests for PHP files are being proxied to our second service, which is called wordpress, on port 9000. The NGINX configuration itself is being mounted from our host machine to /etc/nginx/conf.d/default.conf.

What this means is that our NGINX container is acting as a web server for the static content, the first role, and also as a proxy through to the WordPress container for the dynamic content, which is the second role the container takes on—did you guess right?

The second service is wordpress. This is the official WordPress image from https://hub.docker.com/images/wordpress, and I am using the php7.2-fpm-alpine tag. This gives us a WordPress installation running on PHP 7.2 using PHP-FPM built on top of an Alpine Linux base.

> **Important note**
>
> **PHP FastCGI Process Manager** (**PHP-FPM**) is a PHP FastCGI implementation with some great features. For us, it allows PHP to run as a service that we can bind to a port and pass requests to; this fits in with the Docker method of running a single service on each container.

We are mounting the same web root as we are using for the web service, which on the host machine is wordpress/web and on the service is /var/www/html/. To start off with, the folder on our host machine will be empty; however, once the WordPress service starts, it will detect that there isn't a core WordPress installation and copy one to that location, effectively bootstrapping our WordPress installation and copying it to our host machine, ready for us to start working on it.

The third service is MySQL, which uses the official MySQL image, which can be found at `https://hub.docker.com/images/mysql` and is the only image out of the four we are using that doesn't use Alpine Linux (come on MySQL; pull your finger out and publish an Alpine Linux-based image!). Instead, it uses `debian:buster-slim`.

We are passing a few environment variables so that a database, username, and password are all created when the container first runs; the password is something you should change if you ever use this as a base for one of your projects.

Like the web and `wordpress` containers, we are mounting a folder from our host machine. In this case, it is `wordpress/mysql`, and we are mounting it to `/var/lib/mysql/`, which is the default folder where MySQL stores its databases and associated files.

The fourth and final service is simply called `wp`. It differs from the other three services: this service will immediately exit when executed because there is no long-running process within the container. Instead of a long-running process, we have a single process that is used to interact with and manage our WordPress installation.

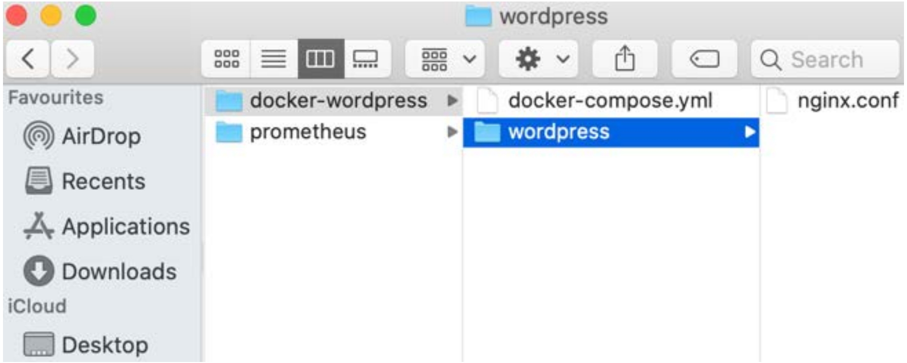
The advantage of running this tool in a container is that the environment we are running the command-line tool in exactly matches our main `wordpress` container.

You will notice that we are mounting the web root as we have done on the web and WordPress, meaning that the container has full access to our WordPress installation as well as a second mount called `/export`; we will look at this in more detail once we have WordPress configured.

To start WordPress, we just need to run the following command to pull the images:

```
$ docker-compose pull
```

This will pull the images and start the web, `wordpress`, and `mysql` services, as well as readying the `wp` service. Before the services start, our `wordpress` folder looks like this:



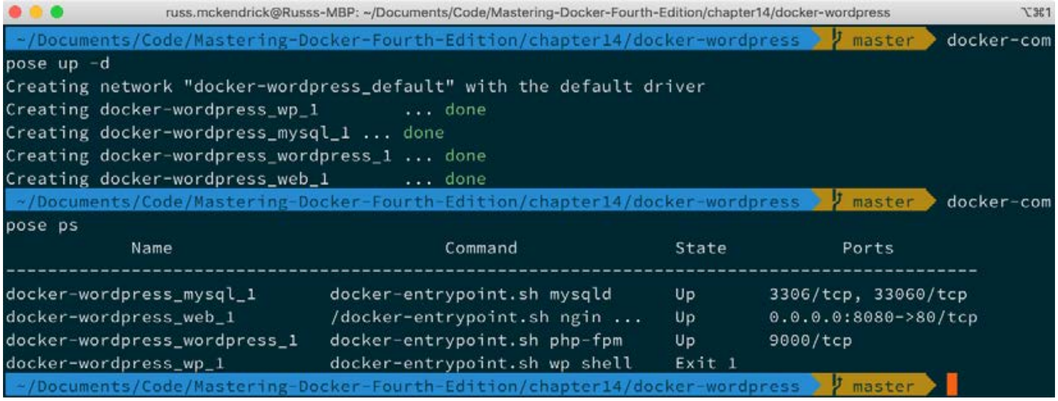
docker-wordpress
prometheus
docker-compose.yml
wordpress
nginx.conf

Figure 15.2: Before launching WordPress

As you can see, we only have **nginx.conf** in there, which is part of the Git repository. Then we can use the following commands to start the containers and check their status:

```
$ docker-compose up -d
$ docker-compose ps
```

Your Terminal output should look similar to the following screen:



```
russ.mckendrick@Russs-MBP: ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress          ⌥⌘1
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  docker-com
pose up -d
Creating network "docker-wordpress_default" with the default driver
Creating docker-wordpress_wp_1        ... done
Creating docker-wordpress_mysql_1 ... done
Creating docker-wordpress_wordpress_1 ... done
Creating docker-wordpress_web_1       ... done
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  docker-com
pose ps
            Name                           Command               State              Ports
-------------------------------------------------------------------------------------------------
docker-wordpress_mysql_1       docker-entrypoint.sh mysqld      Up      3306/tcp, 33060/tcp
docker-wordpress_web_1         /docker-entrypoint.sh ngin ...   Up      0.0.0.0:8080->80/tcp
docker-wordpress_wordpress_1   docker-entrypoint.sh php-fpm     Up      9000/tcp
docker-wordpress_wp_1          docker-entrypoint.sh wp shell    Exit 1
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master
```

Figure 15.3: Launching and checking the status of our WordPress installation

You should see that three folders have been created in the `wordpress` folder: `export`, `mysql`, and `web`. Also, remember that we are expecting `dockerwordpress_wp_1` to have an exit state of `Exit 1`, so that's fine:

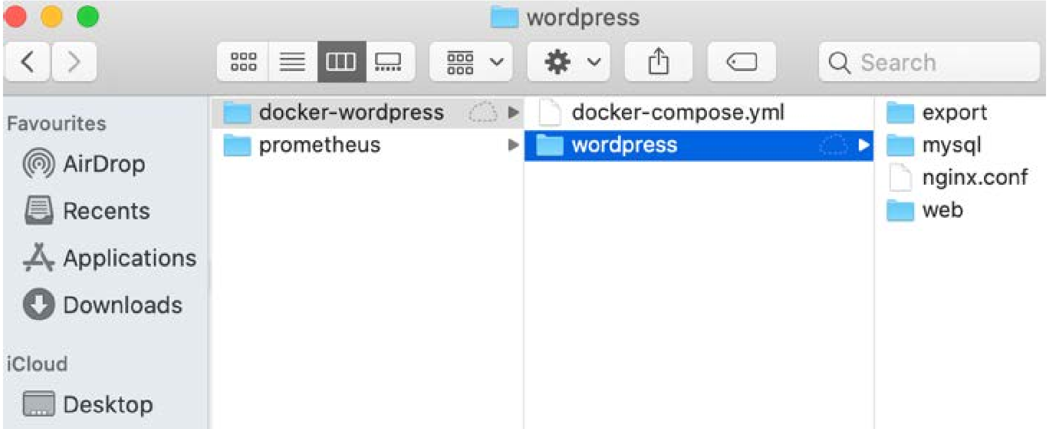


Figure 15.4: Checking the folders created by launching WordPress

Opening a browser and going to `http://localhost:8080/` should show you the standard WordPress preinstallation welcome page, where you can select the language you wish to use for your installation:

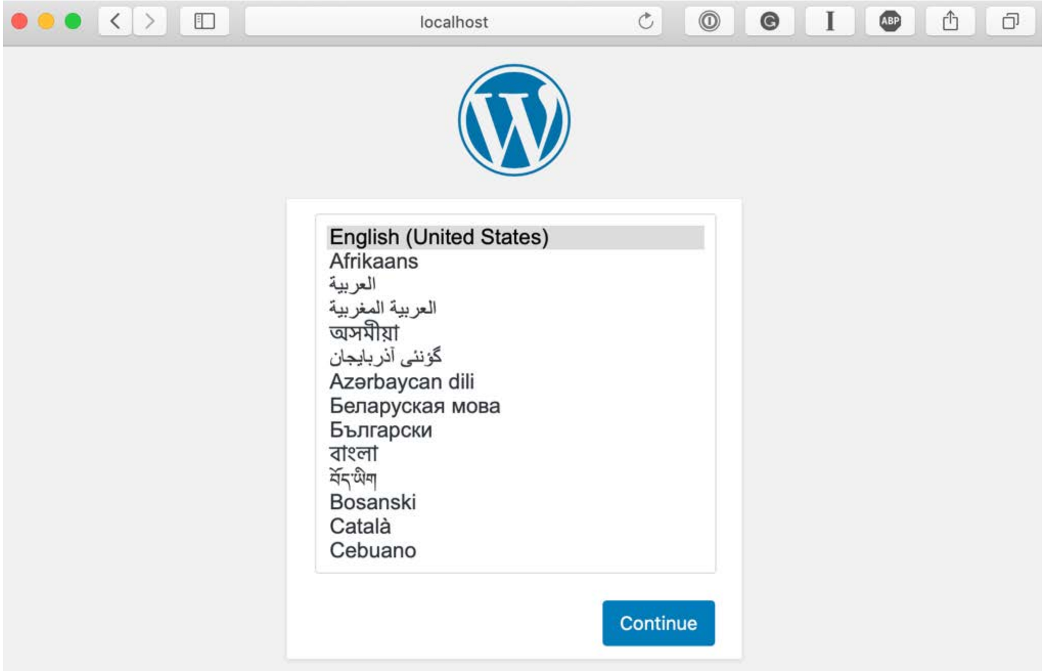


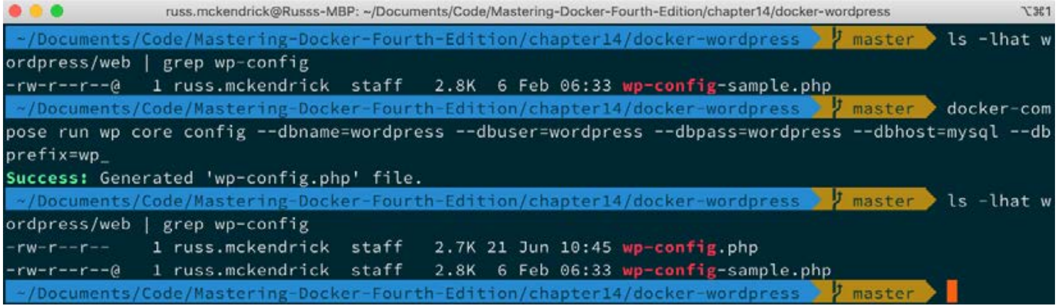
Figure 15.5: The WordPress setup screen

Do not click on **Continue**, as it will take you to the next screen of the GUI-based installation. Instead, return to your Terminal.

Rather than using the GUI to complete the installation, we are going to use `wp-cli`. There are two steps to this. The first step is to create a `wp-config.php` file. To do this, run the following command:

```
$ docker-compose run wp core config \
    --dbname=wordpress \
    --dbuser=wordpress \
    --dbpass=wordpress \
    --dbhost=mysql \
    --dbprefix=wp_
```

As you will see in the following Terminal output, before I ran the command, I just had the `wp-config-sample.php` file, which ships with core WordPress. Then, after running the command, I had my own `wp-config.php` file:


russ.mckendrick@Russs-MBP: ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress
⌥⌘1

```
 ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  ls -lhat w
ordpress/web | grep wp-config
-rw-r--r--@   1 russ.mckendrick  staff   2.8K  6 Feb 06:33 wp-config-sample.php
 ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  docker-com
pose run wp core config --dbname=wordpress --dbuser=wordpress --dbpass=wordpress --dbhost=mysql --db
prefix=wp_
Success: Generated 'wp-config.php' file.
 ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  ls -lhat w
ordpress/web | grep wp-config
-rw-r--r--    1 russ.mckendrick  staff   2.7K 21 Jun 10:45 wp-config.php
-rw-r--r--@   1 russ.mckendrick  staff   2.8K  6 Feb 06:33 wp-config-sample.php
 ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master 
```

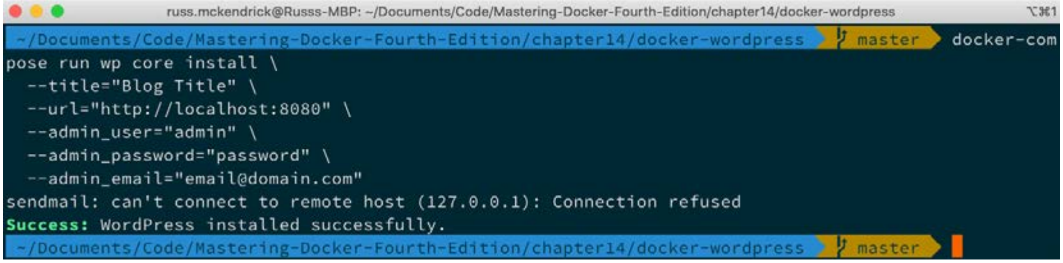
Figure 15.6: Creating the wp-config.php file using wp-cli

You will notice that in the command, we are passing the database details that we defined in the Docker Compose file and telling WordPress that it can connect to the database service at the address of `mysql`.

Now that we have configured database connection details, we need to configure our WordPress site, as well as create an admin user and set a password. To do this, run the following command:

```
$ docker-compose run wp core install \
    --title='Blog Title' \
    --url='http://localhost:8080' \
    --admin_user='admin' \
    --admin_password='password' \
    --admin_email='email@domain.com'
```

Running this command will produce an error in the email service; do not worry about this message, as this is only a local development environment. We are not too worried about emails leaving our WordPress installation:


russ.mckendrick@Russs-MBP: ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress
⌥⌘1

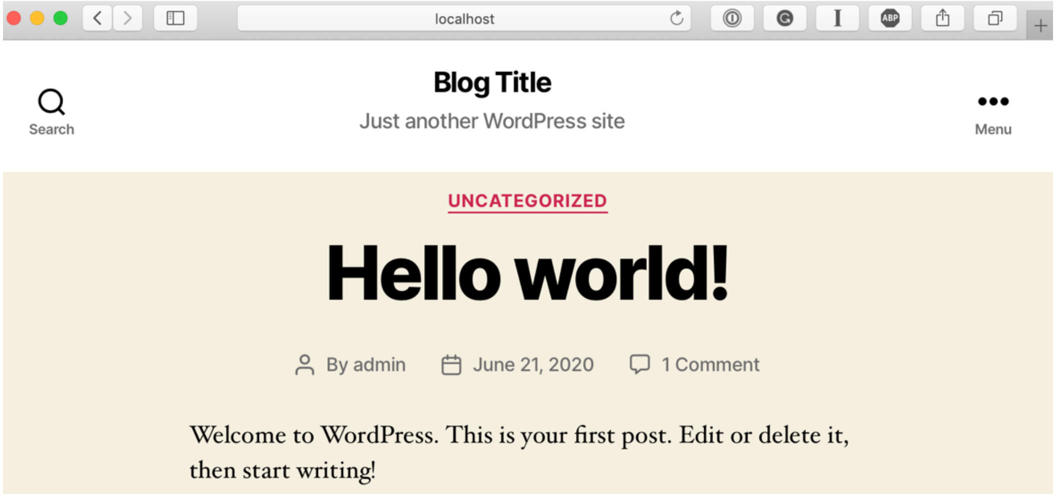
```
 ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  docker-com
pose run wp core install \
  --title="Blog Title" \
  --url="http://localhost:8080" \
  --admin_user="admin" \
  --admin_password="password" \
  --admin_email="email@domain.com"
sendmail: can't connect to remote host (127.0.0.1): Connection refused
Success: WordPress installed successfully.
 ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master 
```

Figure 15.7: Configuring WordPress using cp-cli

We have used wp-cli to configure the following in WordPress:

- Our URL is **http://localhost:8080**.
- Our site title should be **Blog Title**.
- Our admin username is **admin** and our password is **password**, and the user has the email address **email@domain.com**.

Going back to your browser and entering `http://localhost:8080/` should present you with a vanilla WordPress site:



Blog Title

Just another WordPress site

UNCATEGORIZED

Hello world!

By admin June 21, 2020 1 Comment

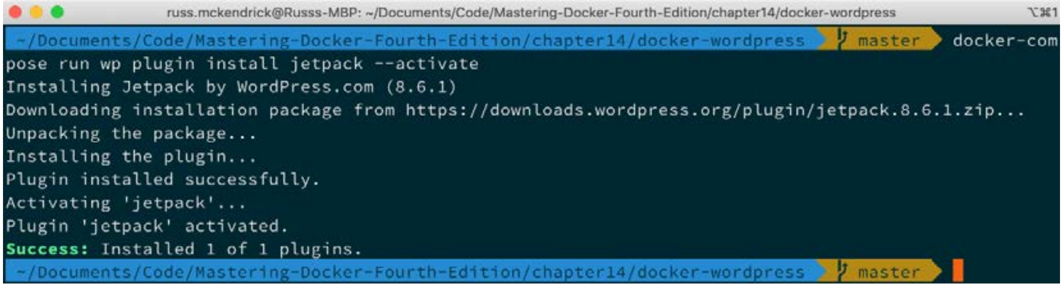
Welcome to WordPress. This is your first post. Edit or delete it, then start writing!

Figure 15.8: A default WordPress site

Before we do anything further, let's customize our installation a little, first by installing and enabling the **JetPack** plugin:

```
$ docker-compose run wp plugin install jetpack -activate
```

The output of the command is given here:


russ.mckendrick@Russs-MBP: ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress

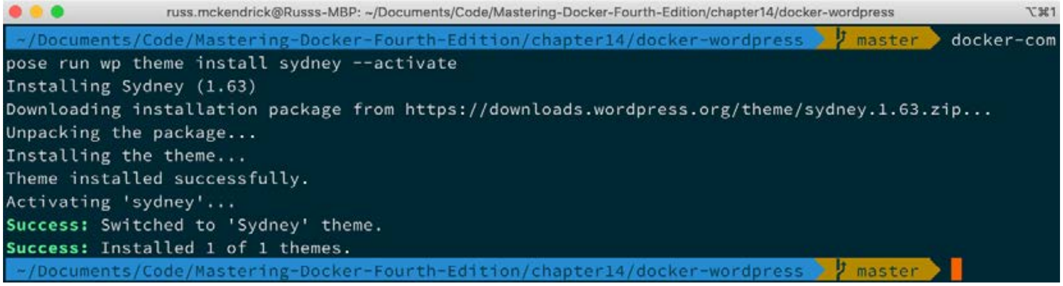
```
 ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  docker-com
pose run wp plugin install jetpack --activate
Installing Jetpack by WordPress.com (8.6.1)
Downloading installation package from https://downloads.wordpress.org/plugin/jetpack.8.6.1.zip...
Unpacking the package...
Installing the plugin...
Plugin installed successfully.
Activating 'jetpack'...
Plugin 'jetpack' activated.
Success: Installed 1 of 1 plugins.
 ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master 
```

Figure 15.9: Installing the JetPack plugin

Then, `install` and enable the **Sydney** theme:

```
$ docker-compose run wp theme install sydney --activate
```

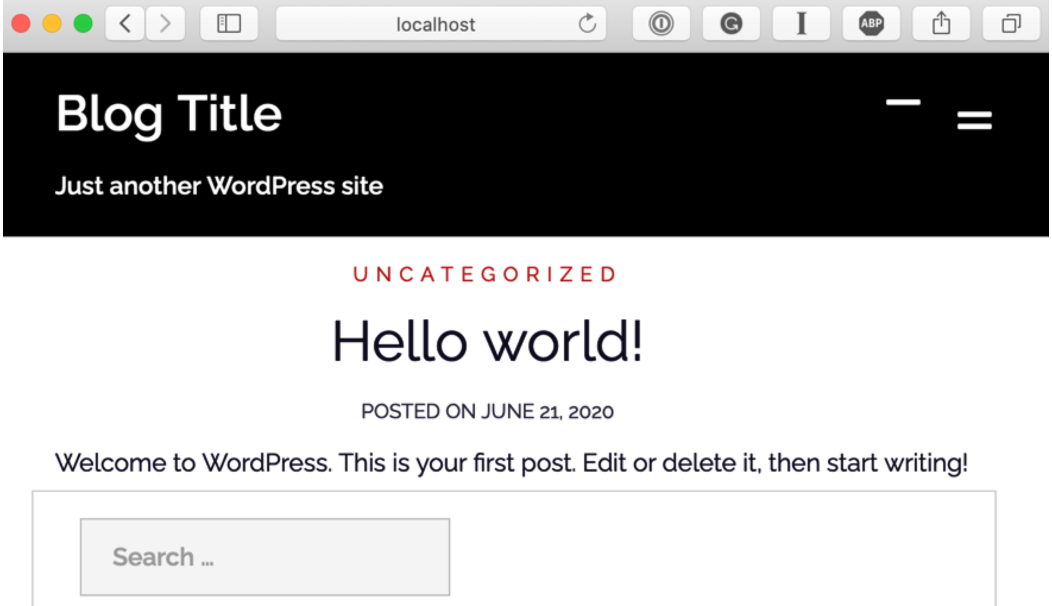
The output of the command is given here:



```
russ.mckendrick@Russs-MBP: ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress    ⌥⌘1
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  docker-compose run wp theme install sydney --activate
Installing Sydney (1.63)
Downloading installation package from https://downloads.wordpress.org/theme/sydney.1.63.zip...
Unpacking the package...
Installing the theme...
Theme installed successfully.
Activating 'sydney'...
Success: Switched to 'Sydney' theme.
Success: Installed 1 of 1 themes.
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master 
```

Figure 15.10: Installing the Sydney theme

Refreshing our WordPress page at `http://localhost:8080/` should show something like the following:



Blog Title

Just another WordPress site

UNCATEGORIZED

Hello world!

POSTED ON JUNE 21, 2020

Welcome to WordPress. This is your first post. Edit or delete it, then start writing!

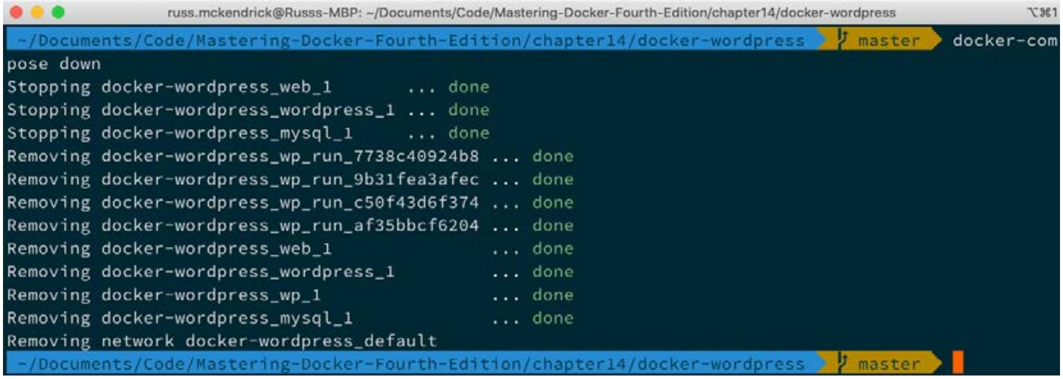
Search ...

Figure 15.11: Viewing our site with the updated theme

Before we open our IDE, let's destroy the containers running our WordPress installation using the following command:

```
$ docker-compose down
```

The output of the command is given here:



```
russ.mckendrick@Russs-MBP: ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress   ⌥⌘1
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  docker-com
pose down
Stopping docker-wordpress_web_1       ... done
Stopping docker-wordpress_wordpress_1 ... done
Stopping docker-wordpress_mysql_1     ... done
Removing docker-wordpress_wp_run_7738c40924b8 ... done
Removing docker-wordpress_wp_run_9b31fea3afec ... done
Removing docker-wordpress_wp_run_c50f43d6f374 ... done
Removing docker-wordpress_wp_run_af35bbcf6204 ... done
Removing docker-wordpress_web_1               ... done
Removing docker-wordpress_wordpress_1         ... done
Removing docker-wordpress_wp_1                ... done
Removing docker-wordpress_mysql_1             ... done
Removing network docker-wordpress_default
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master
```

Figure 15.12: Stopping and removing the containers that are running WordPress

As our entire WordPress installation, including all of the files and the database, is stored on our local machine, we should be able to run the following command to return our WordPress site where we left it:

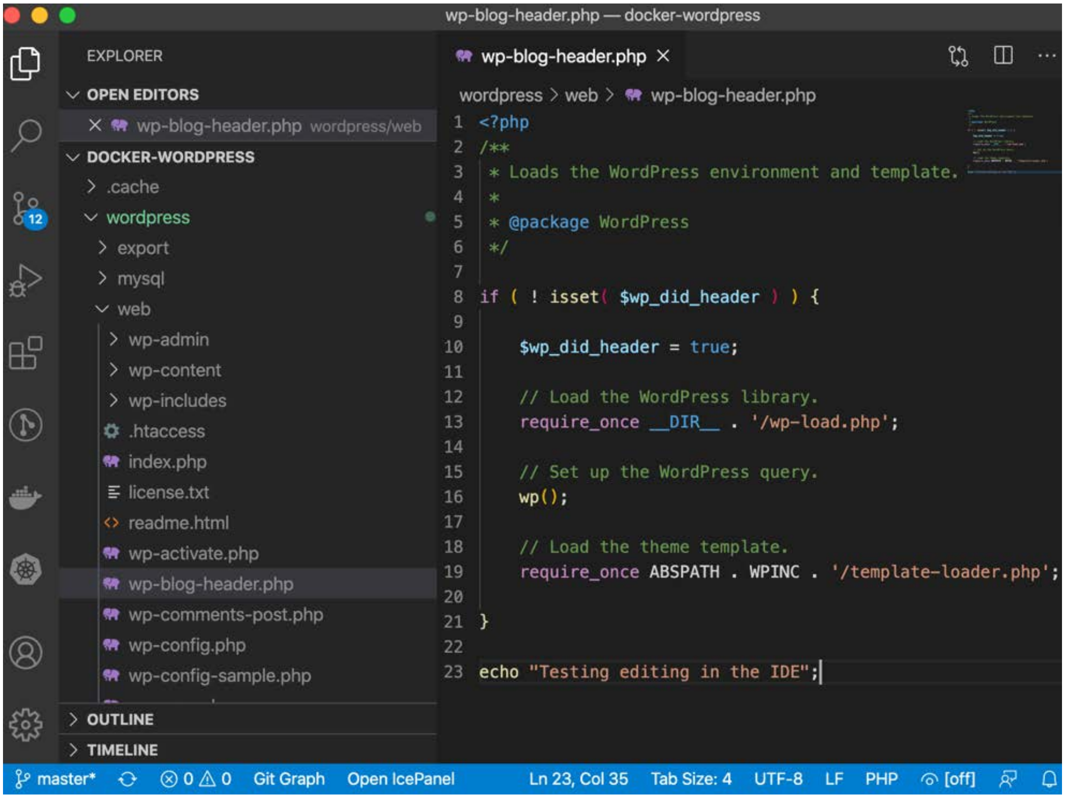
```
$ docker-compose up -d
```

Once you have confirmed that it is up and running as expected by going to `http://localhost:8080/`, open the `docker-wordpress` folder in your desktop editor. I used Visual Studio Code.

In your editor, open the `wordpress/web/wp-blog-header.php` file, add the following line to the opening PHP statement, and save it:

```
echo 'Testing editing in the IDE';
```

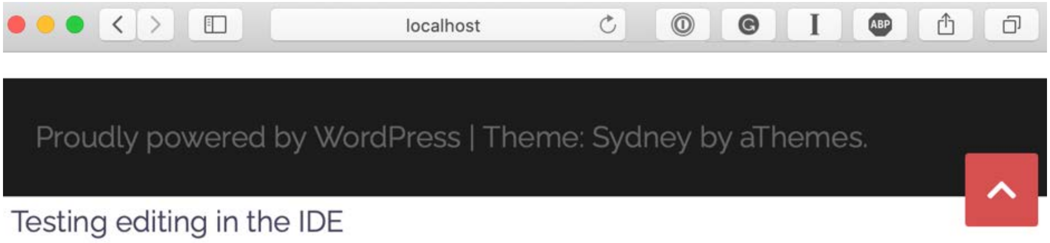
The file should look something like the following:



```php
<?php
/**
 * Loads the WordPress environment and template.
 *
 * @package WordPress
 */

if ( ! isset( $wp_did_header ) ) {

    $wp_did_header = true;

    // Load the WordPress library.
    require_once __DIR__ . '/wp-load.php';

    // Set up the WordPress query.
    wp();

    // Load the theme template.
    require_once ABSPATH . WPINC . '/template-loader.php';

}

echo "Testing editing in the IDE";
```

Figure 15. 13: Editing wp-blog-header.php in Visual Studio Code

Once saved, refresh your browser. You should see the message **Testing editing in the IDE** at the very bottom of the page (the following screen is zoomed; it may be more difficult to spot if you are following along on screen, as the text is quite small):



Proudly powered by WordPress | Theme: Sydney by aThemes.

Testing editing in the IDE

Figure 15.14: Viewing our edit on the page

The final thing we are going to look at is why we had the `wordpress/export` folder mounted on the `wp` container.

As already mentioned earlier in the chapter, you shouldn't really be touching the contents of the `wordpress/mysql` folder; this also includes sharing it. While it would probably work if you were to zip up your project folder and pass it to a colleague, this is not considered best practice. Because of this, we have mounted the export folder to allow us to use WPCLI to make a database dump and import it.

To do this, run the following command:

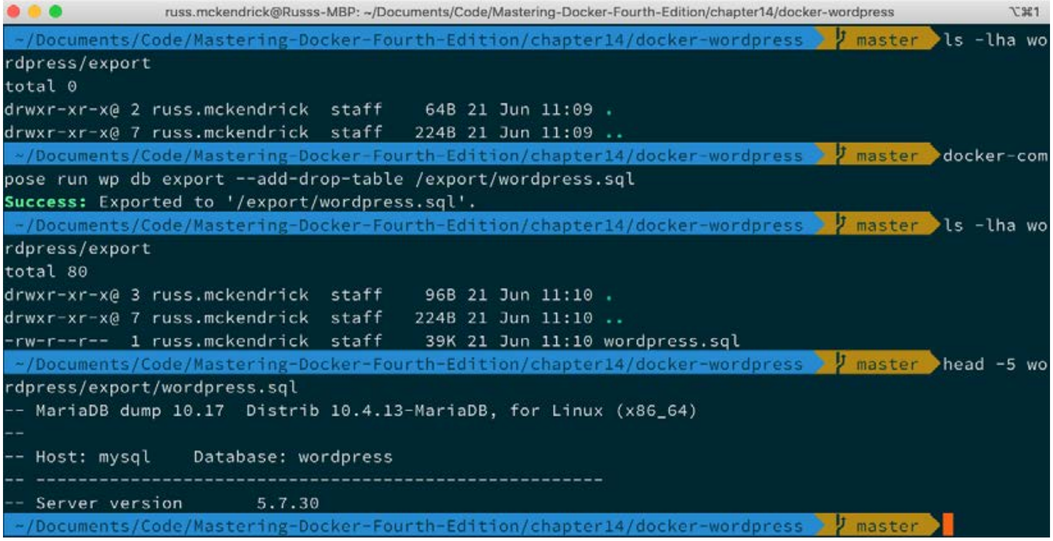
```
$ docker-compose run wp db export --add-drop-table /export/
wordpress.sql
```

Depending on the version of Docker you are running, you may receive a permission-denied error when running the preceding command; if you do, then run the following command instead:

```
$ docker-compose run wp db export --add-drop-table /var/www/
html/wordpress.sql
```

This will copy the database dump to `wordpress/wordpress` rather than `wordpress/export`. The reason for this is that different host operating systems handle the creation of local files differently, which can cause permission issues within the container itself.

The following Terminal output shows the export and also the contents of `wordpress/export` before and after the command being run, and lastly, the top few lines of the MySQL dump file:



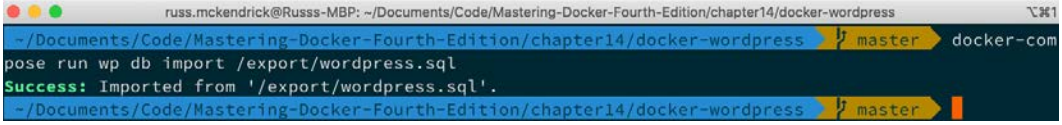
```
russ.mckendrick@Russs-MBP: ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress ⌥⌘1
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master ls -lha wo
rdpress/export
total 0
drwxr-xr-x@ 2 russ.mckendrick  staff    64B 21 Jun 11:09 .
drwxr-xr-x@ 7 russ.mckendrick  staff   224B 21 Jun 11:09 ..
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master docker-com
pose run wp db export --add-drop-table /export/wordpress.sql
Success: Exported to '/export/wordpress.sql'.
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master ls -lha wo
rdpress/export
total 80
drwxr-xr-x@ 3 russ.mckendrick  staff    96B 21 Jun 11:10 .
drwxr-xr-x@ 7 russ.mckendrick  staff   224B 21 Jun 11:10 ..
-rw-r--r--  1 russ.mckendrick  staff    39K 21 Jun 11:10 wordpress.sql
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master head -5 wo
rdpress/export/wordpress.sql
-- MariaDB dump 10.17  Distrib 10.4.13-MariaDB, for Linux (x86_64)
--
-- Host: mysql    Database: wordpress
-- ------------------------------------------------------
-- Server version       5.7.30
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master
```

Figure 15.15: Dumping the WordPress database

If I had, say, made a mistake during development and accidently trashed part of my database, I could roll back to the backup of the database I made by running the following command:

```
$ docker-compose run wp db import /export/wordpress.sql
```

The output of the command is given here:



```
russ.mckendrick@Russs-MBP: ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress ⌥⌘1
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master  docker-com
pose run wp db import /export/wordpress.sql
Success: Imported from '/export/wordpress.sql'.
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master
```

Figure 15.16: Importing the WordPress database

As you can seen, we have installed WordPress, interacted with it using both the WordPress command-line tool, `wp-cli`, and also in a web browser, edited the code, and backed up and restored the database, all without having to install or configure `NGINX`, `PHP`, `MySQL`, or `wp-cli` on our local machine. Nor did we have to log in to a container. By mounting volumes from our host machine, our content was safe when we tore our WordPress containers down and we didn't lose any work.

Also, if needed, we could have easily passed a copy of our project folder to a colleague who has Docker installed, and with a single command, they could be working on our code, knowing that it is running in the same exact environment as our own installation.

Tip

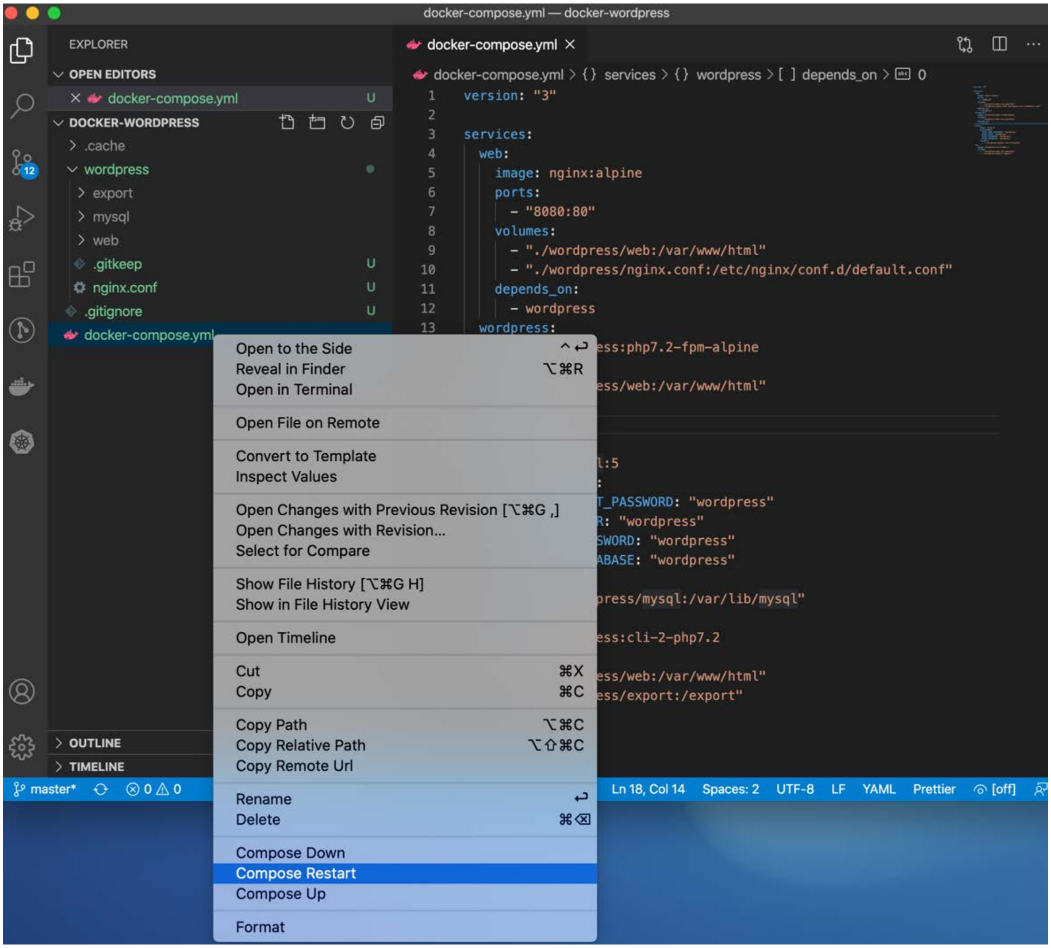
If you like, you can stop and remove your WordPress containers by running `docker-compose` down. If you are following along, you might want to keep WordPress for the next section so that you have running containers to monitor.

Finally, as we're using official images from the Docker Hub, we know we can safely ask to have them deployed into production, as they have been built with Docker's best practices in mind.

One thing that you may find really useful is how well Docker can be integrated in your IDE of choice. A few pages back, when we edited the `wp-blog-header.php` file, you may have noticed a Docker icon on the left-hand side of the screen. Before we finish this section of the chapter, let's quickly discuss how Microsoft have integrated Docker support into Visual Studio Code, or VS Code, as we will be calling it from now on.

The first thing you need to do is install VS Code, which you can find at `https://code.visualstudio.com/`, and the Microsoft Docker extension, which can found in the Visual Studio Market place at `https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker`.

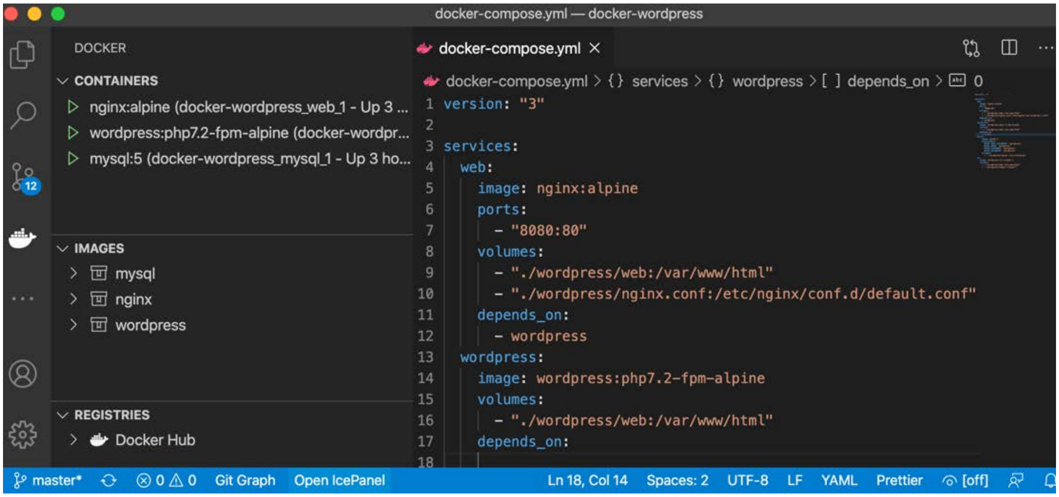
You will notice that the extension is not in your face. This is because it integrates quite deeply with VS Code—for example, if you were to right-click over the `docker-compose.yml` file in the explorer, you will notice that the menu has some options that allow you to interact with Docker Compose:



```
version: "3"

services:
  web:
    image: nginx:alpine
    ports:
      - "8080:80"
    volumes:
      - "./wordpress/web:/var/www/html"
      - "./wordpress/nginx.conf:/etc/nginx/conf.d/default.conf"
    depends_on:
      - wordpress
  wordpress:
ess:php7.2-fpm-alpine

ess/web:/var/www/html"


l:5
:
T_PASSWORD: "wordpress"
R: "wordpress"
SWORD: "wordpress"
ABASE: "wordpress"

press/mysql:/var/lib/mysql"

ess:cli-2-php7.2

ess/web:/var/www/html"
ess/export:/export"
```

Figure 15.17: Running Docker Compose from VS Code

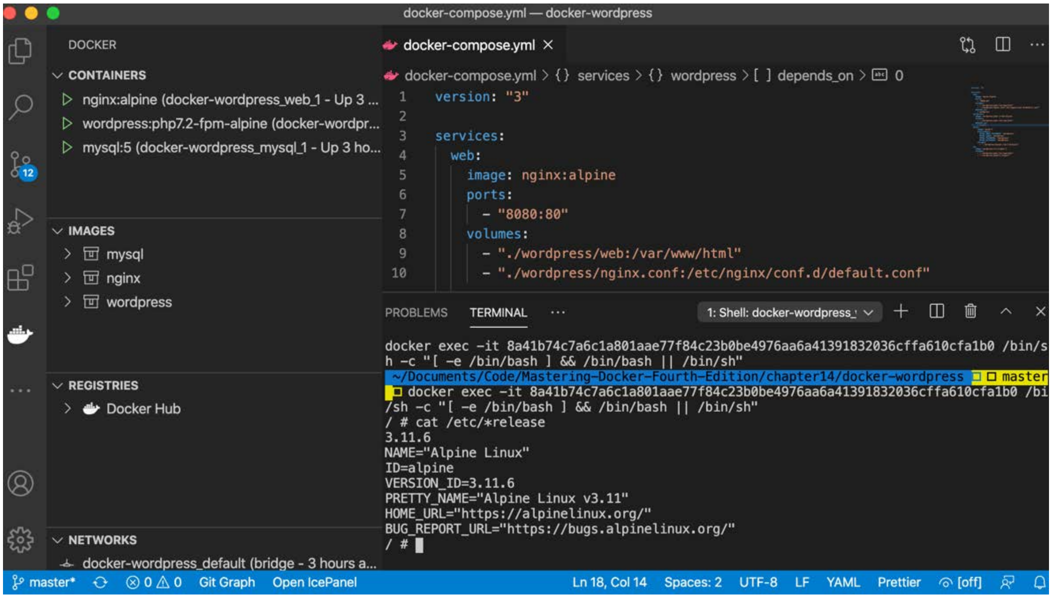
Clicking on the **Docker** icon on the left-hand side will bring up a list running Containers, available Images, Registries that you are connected to, Networks, and Volumes:


docker-compose.yml — docker-wordpress
DOCKER
CONTAINERS
nginx:alpine (docker-wordpress_web_1 - Up 3 ...
wordpress:php7.2-fpm-alpine (docker-wordpr...
mysql:5 (docker-wordpress_mysql_1 - Up 3 ho...
IMAGES
mysql
nginx
wordpress
REGISTRIES
Docker Hub
docker-compose.yml
docker-compose.yml > {} services > {} wordpress > [ ] depends_on > 0
master* ⊗ 0 ⚠ 0 Git Graph Open IcePanel
Ln 18, Col 14 Spaces: 2 UTF-8 LF YAML Prettier [off]

```
version: "3"

services:
  web:
    image: nginx:alpine
    ports:
      - "8080:80"
    volumes:
      - "./wordpress/web:/var/www/html"
      - "./wordpress/nginx.conf:/etc/nginx/conf.d/default.conf"
    depends_on:
      - wordpress
  wordpress:
    image: wordpress:php7.2-fpm-alpine
    volumes:
      - "./wordpress/web:/var/www/html"
    depends_on:

```

Figure 15.18: Viewing your running containers

Right-clicking over a container gives you the option to attach to the running container using the terminal that is built into VS Code:

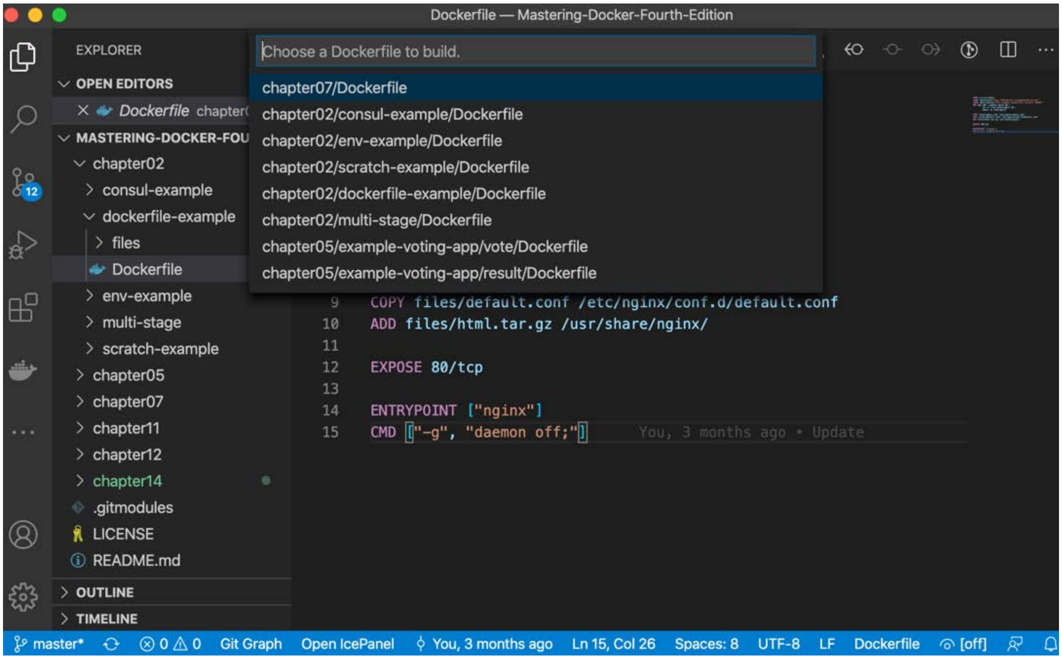

docker-compose.yml — docker-wordpress
DOCKER
CONTAINERS
nginx:alpine (docker-wordpress_web_1 - Up 3 ...
wordpress:php7.2-fpm-alpine (docker-wordpr...
mysql:5 (docker-wordpress_mysql_1 - Up 3 ho...
IMAGES
mysql
nginx
wordpress
REGISTRIES
Docker Hub
NETWORKS
docker-wordpress_default (bridge - 3 hours a...
docker-compose.yml
docker-compose.yml > {} services > {} wordpress > [ ] depends_on > 0
PROBLEMS
TERMINAL
1: Shell: docker-wordpress_
master* ⊗ 0 ⚠ 0 Git Graph Open IcePanel
Ln 18, Col 14 Spaces: 2 UTF-8 LF YAML Prettier [off]

```
version: "3"

services:
  web:
    image: nginx:alpine
    ports:
      - "8080:80"
    volumes:
      - "./wordpress/web:/var/www/html"
      - "./wordpress/nginx.conf:/etc/nginx/conf.d/default.conf"
```

```
docker exec -it 8a41b74c7a6c1a801aae77f84c23b0be4976aa6a41391832036cffa610cfa1b0 /bin/s
h -c "[ -e /bin/bash ] && /bin/bash || /bin/sh"
 ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/docker-wordpress  master
 docker exec -it 8a41b74c7a6c1a801aae77f84c23b0be4976aa6a41391832036cffa610cfa1b0 /bi
/sh -c "[ -e /bin/bash ] && /bin/bash || /bin/sh"
/ # cat /etc/*release
3.11.6
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.11.6
PRETTY_NAME="Alpine Linux v3.11"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
/ #
```

Figure 15.19: Attaching to a container using the built-in terminal

Opening the folder from the Git repository that accompanies this title in VS Code and then pressing *CMD* +*Shift* + *P* will open the command prompt in VS Code. From here, type **Build** and select **Docker Images: Build**. VS Code will then scan the entire repository for `Dockerfile` files and ask you which one you want to build:



Dockerfile — Mastering-Docker-Fourth-Edition

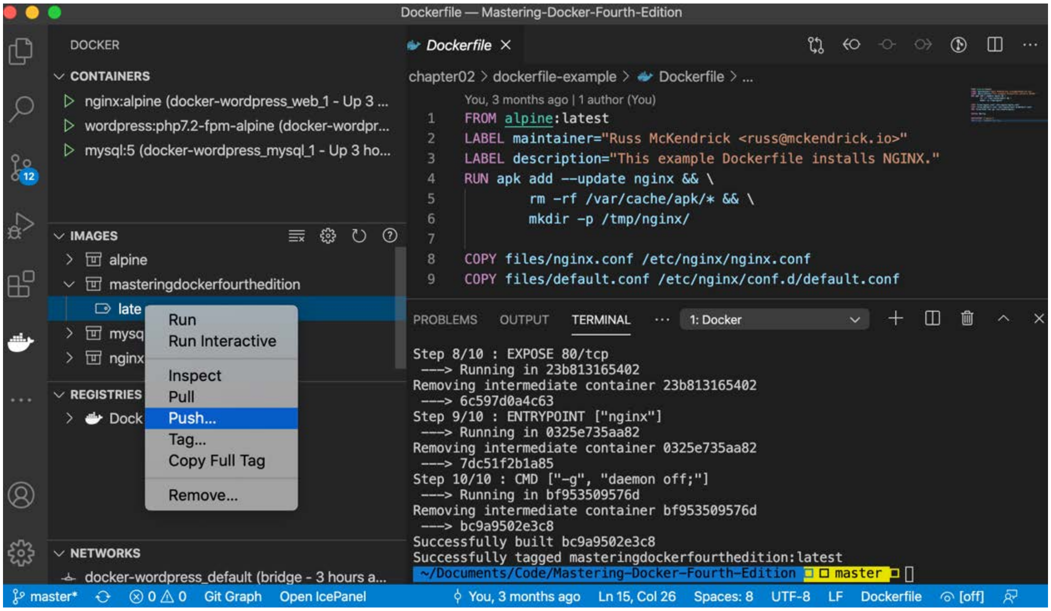
Choose a Dockerfile to build.

- chapter07/Dockerfile
- chapter02/consul-example/Dockerfile
- chapter02/env-example/Dockerfile
- chapter02/scratch-example/Dockerfile
- chapter02/dockerfile-example/Dockerfile
- chapter02/multi-stage/Dockerfile
- chapter05/example-voting-app/vote/Dockerfile
- chapter05/example-voting-app/result/Dockerfile

```
COPY files/default.conf /etc/nginx/conf.d/default.conf
ADD files/html.tar.gz /usr/share/nginx/

EXPOSE 80/tcp

ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]     You, 3 months ago • Update
```

Figure 15.20: Choosing a Dockerfile to build

Once built, your image will be listed in the Docker section, and you can right-click over the tag to push it to any of the registries you are connected to:

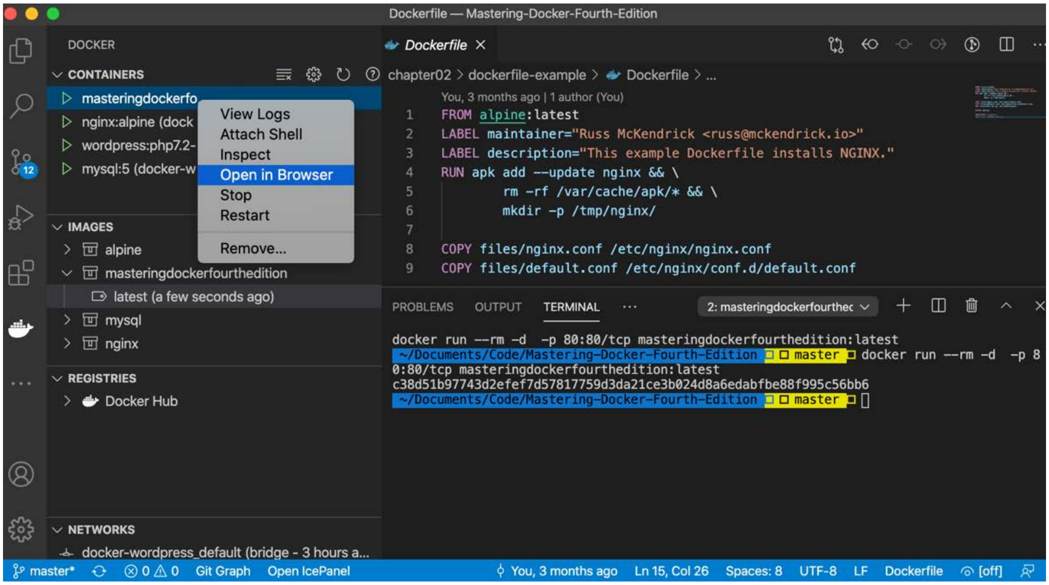

Dockerfile — Mastering-Docker-Fourth-Edition
DOCKER
CONTAINERS
nginx:alpine (docker-wordpress_web_1 - Up 3 ...
wordpress:php7.2-fpm-alpine (docker-wordpr...
mysql:5 (docker-wordpress_mysql_1 - Up 3 ho...
IMAGES
alpine
masteringdockerfourthedition
late
mysq
nginx
REGISTRIES
Dock
NETWORKS
docker-wordpress_default (bridge - 3 hours a...
Run
Run Interactive
Inspect
Pull
Push...
Tag...
Copy Full Tag
Remove...
Dockerfile
chapter02 > dockerfile-example > Dockerfile > ...
You, 3 months ago | 1 author (You)
PROBLEMS
OUTPUT
TERMINAL
1: Docker
master*
0 0
Git Graph
Open IcePanel
You, 3 months ago
Ln 15, Col 26
Spaces: 8
UTF-8
LF
Dockerfile
[off]

```
FROM alpine:latest
LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
LABEL description="This example Dockerfile installs NGINX."
RUN apk add --update nginx && \
        rm -rf /var/cache/apk/* && \
        mkdir -p /tmp/nginx/

COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/default.conf /etc/nginx/conf.d/default.conf
```

```
Step 8/10 : EXPOSE 80/tcp
 ---> Running in 23b813165402
Removing intermediate container 23b813165402
 ---> 6c597d0a4c63
Step 9/10 : ENTRYPOINT ["nginx"]
 ---> Running in 0325e735aa82
Removing intermediate container 0325e735aa82
 ---> 7dc51f2b1a85
Step 10/10 : CMD ["-g", "daemon off;"]
 ---> Running in bf953509576d
Removing intermediate container bf953509576d
 ---> bc9a9502e3c8
Successfully built bc9a9502e3c8
Successfully tagged masteringdockerfourthedition:latest
 ~/Documents/Code/Mastering-Docker-Fourth-Edition  master
```

Figure 15.21: Pushing our newly built image

You can also run the container. Once it is running, you can then right-click on it and select **Open in browser** to go straight to the application:



Dockerfile — Mastering-Docker-Fourth-Edition
DOCKER
CONTAINERS
masteringdockerfo...
nginx:alpine (dock
wordpress:php7.2-
mysql:5 (docker-w
View Logs
Attach Shell
Inspect
Open in Browser
Stop
Restart
Remove...
IMAGES
alpine
masteringdockerfourthedition
latest (a few seconds ago)
mysql
nginx
REGISTRIES
Docker Hub
NETWORKS
docker-wordpress_default (bridge - 3 hours a...
Dockerfile
chapter02 > dockerfile-example > Dockerfile > ...
You, 3 months ago | 1 author (You)
PROBLEMS
OUTPUT
TERMINAL
2: masteringdockerfourthec
master* 0 0 Git Graph Open IcePanel
You, 3 months ago Ln 15, Col 26 Spaces: 8 UTF-8 LF Dockerfile [off]

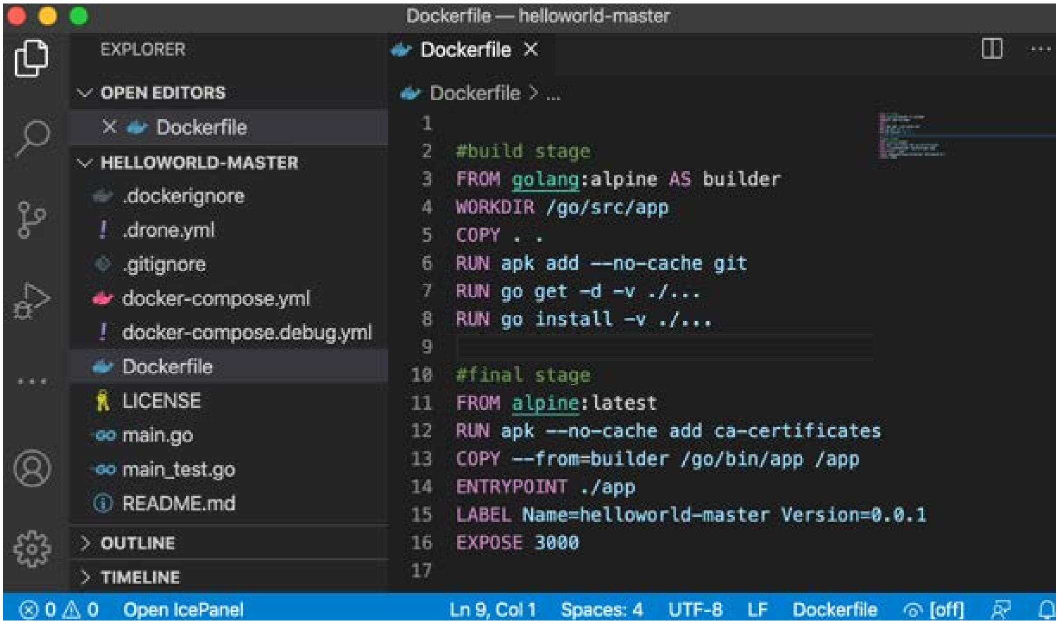
```
FROM alpine:latest
LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
LABEL description="This example Dockerfile installs NGINX."
RUN apk add --update nginx && \
        rm -rf /var/cache/apk/* && \
        mkdir -p /tmp/nginx/

COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/default.conf /etc/nginx/conf.d/default.conf
```

```
docker run --rm -d  -p 80:80/tcp masteringdockerfourthedition:latest
 ~/Documents/Code/Mastering-Docker-Fourth-Edition  master  docker run --rm -d  -p 8
0:80/tcp masteringdockerfourthedition:latest
c38d51b97743d2efef7d57817759d3da21ce3b024d8a6edabfbe88f995c56bb6
 ~/Documents/Code/Mastering-Docker-Fourth-Edition  master 
```

Figure 15.22: Opening a running container in your browser

The final trick up the VS Code Docker extension's sleeve that we are going to cover is an extremely useful one. Let's say that you have a repository with no `Dockerfile`—for example, the Go Training `helloworld` repository, which can be found at `https://github.com/go-training/helloworld/`, has no `Dockerfile` or `docker-compose.yml` files. Grab a copy of it and open it in VS Code.

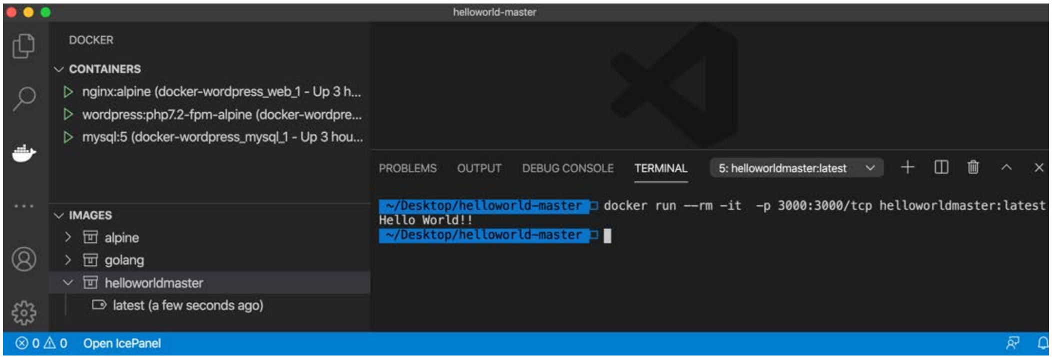
Once open, press *CMD +Shift + P*, type in `Add Docker`, and then select **Docker: Add Dockerfiles to workspace**. You will be asked to select a platform. We know that the code in the workspace is Go, so select that from the list. You will then be asked what ports you want to expose. Leave it at the default `3000`. Once you hit **Return**, a `Dockerfile`, which looks like the following, will be opened:



```
#build stage
FROM golang:alpine AS builder
WORKDIR /go/src/app
COPY . .
RUN apk add --no-cache git
RUN go get -d -v ./...
RUN go install -v ./...

#final stage
FROM alpine:latest
RUN apk --no-cache add ca-certificates
COPY --from=builder /go/bin/app /app
ENTRYPOINT ./app
LABEL Name=helloworld-master Version=0.0.1
EXPOSE 3000
```

Figure 15.23: A VS-Code-generated, multistage Dockerfile

You will also notice that a `docker-compose.yml` file along with a `.dockerignore` and a few other files have been generated. From here, you can build the image, and then run it. I recommend using the **Run interactive** option as all the application does is print **Hello World!** and then exit, as shown in the following screenshot:


helloworld-master
DOCKER
CONTAINERS
nginx:alpine (docker-wordpress_web_1 - Up 3 h...
wordpress:php7.2-fpm-alpine (docker-wordpre...
mysql:5 (docker-wordpress_mysql_1 - Up 3 hou...
IMAGES
alpine
golang
helloworldmaster
latest (a few seconds ago)
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
5: helloworldmaster:latest
⊗ 0 ⚠ 0 Open IcePanel

```
~/Desktop/helloworld-master  docker run --rm -it  -p 3000:3000/tcp helloworldmaster:latest
Hello World!!
~/Desktop/helloworld-master
```

Figure 15.24: Running the application

As I am sure you have seen, the Docker integration with VS Code is extremely powerful, and enables you to run pretty much every Docker command that we have covered in previous chapters from within VS Code. There are similar extensions for other IDEs. These are linked in the *Further reading* section.

Docker and Azure DevOps

In *Chapter 3, Storing and Distributing Images* in the *Reviewing third-party registries* section, we looked at how we can use GitHub to both host and also build our container images. We also discussed Azure Container Registry.

To close this section of the chapter, we are going to quickly look at getting an Azure DevOps pipeline configured that builds the multistage Dockerfile that we covered in *Chapter 2, Building Container Images*.

Before we configure our pipeline, let's discuss what Azure DevOps is. It is a service offered by Microsoft that provides the following capabilities:

- Version control
- Reporting
- Requirements management
- Project management
- Automated builds
- Testing
- Release management

That might seem like a lot of different services, and it is, but Azure DevOps is the glue that can bind together various Microsoft services in both the Microsoft Azure ecosystem and their programming languages, such as .NET, and tools, such as Visual Studio. Covering everything would take up an entire book; in fact, there are several on the subject, so we will only be touching upon the basic functionality needed to build our container and push it to the Docker Hub.

The only requirement you need to get started with Azure DevOps is an account—to sign up for free, go to `https://dev.azure.com/` and follow the on-screen prompts. Once you have created your account, click on the **+ New project** button.

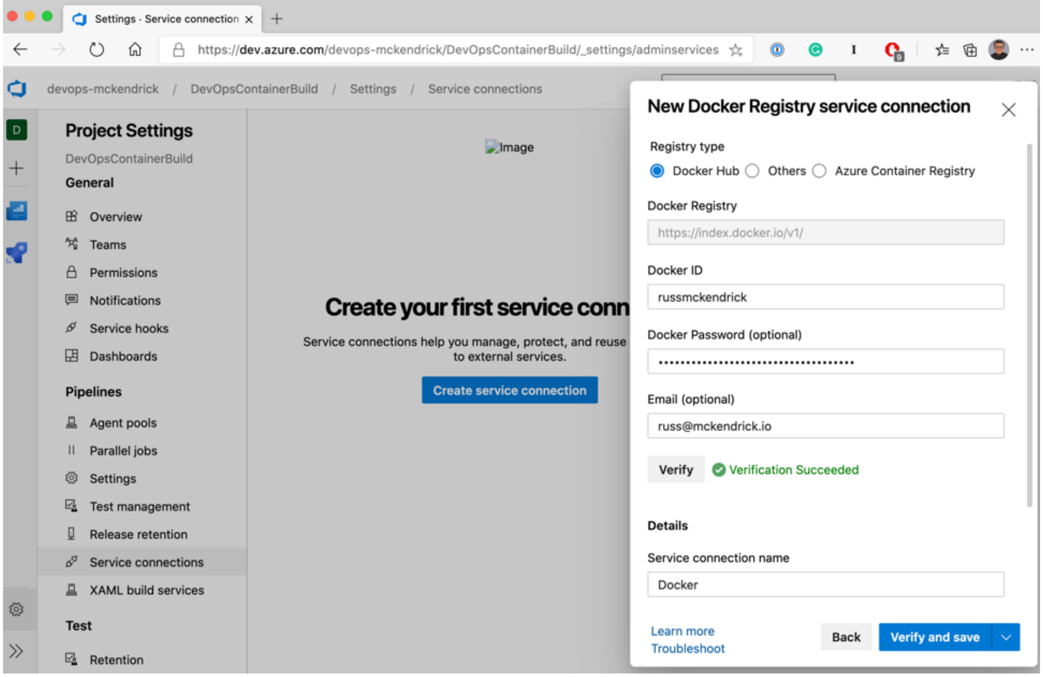
Once you are on the **Create new project** page, you will find the option to fill in a **Project Name** and **Description** and choose the **Visibility**; by default, projects are **Private**, but you can also make them **Public**.

Fill in the details and then click on **Create**. I would recommend making your project **Private**.

Once your project has been created, click on the **Project Settings** option, which can be found at the very bottom of the left-hand side menu. Once the **Project Settings** page loads, click on **Service connections**, which can be found under **Pipelines**.

From there, click the **Create Service** connection button and select **Docker Registry** from the list of services you are presented with.

From here, select the radio icon next to **Docker Hub**, enter your Docker ID, and then enter your **Docker password**. If your Docker Hub account is protected by multifactor authentication, which I really recommend you configure, then you will need a user access token—we covered this in the Docker Hub section of *Chapter 3, Storing and Distributing Images*:



New Docker Registry service connection

Registry type

● Docker Hub ○ Others ○ Azure Container Registry

Docker Registry: https://index.docker.io/v1/

Docker ID: russmckendrick

Docker Password (optional): ••••••••••••••••••••••••••••••••••••

Email (optional): russ@mckendrick.io

Verify ✓ Verification Succeeded

Details

Service connection name: Docker

Settings - Service connection
https://dev.azure.com/devops-mckendrick/DevOpsContainerBuild/_settings/adminservices
devops-mckendrick / DevOpsContainerBuild / Settings / Service connections
Project Settings
DevOpsContainerBuild
General
Overview
Teams
Permissions
Notifications
Service hooks
Dashboards
Pipelines
Agent pools
Parallel jobs
Settings
Test management
Release retention
Service connections
XAML build services
Test
Retention
Image
Create your first service conn
Service connections help you manage, protect, and reuse to external services.
Create service connection
Learn more
Troubleshoot
Back
Verify and save

Figure 15.25: Setting up the service connection to Docker Hub

Once you have entered your details, click on the **Verify** button, and if the details you entered are correct, you will get a green tick. Before you click on the **Verify and save** button, you need to enter a **Service connection name**; I entered **Docker**, but you can use whatever you like—just make a note of it as we will need it shortly.

Next up, you will need a Git repository that contains the `Dockerfile` as well as a file called `azure-pipelines.yml`—there is an example repository at `https://github.com/russmckendrick/DevOpsContainerBuild` that you can fork.

Once you have your repository, return to your Azure DevOps project and then click on **Pipelines** in the left-hand side menu—here, you will be presented with the following screen:

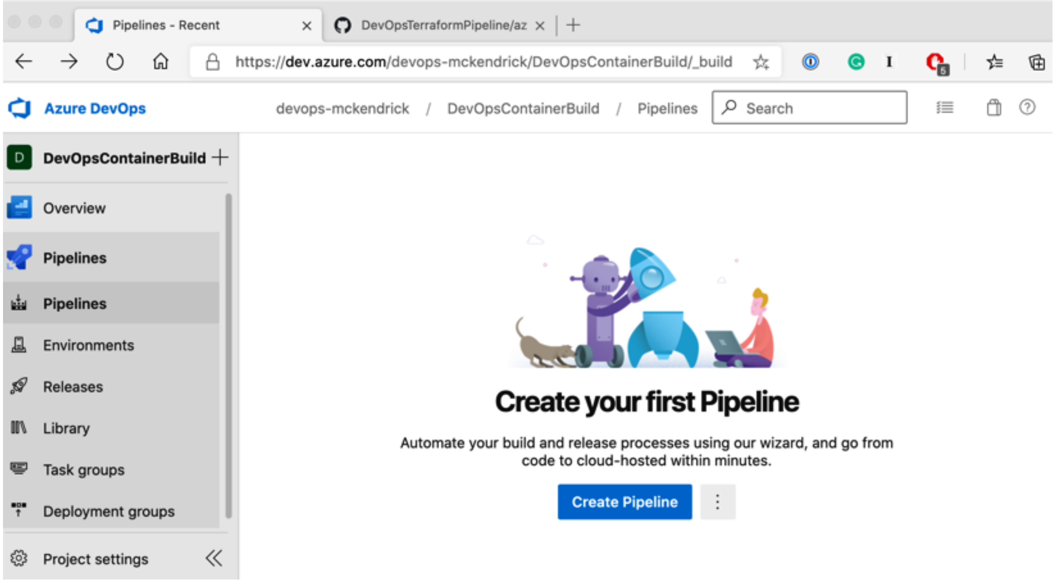

Pipelines - Recent
DevOpsTerraformPipeline/az
https://dev.azure.com/devops-mckendrick/DevOpsContainerBuild/_build
Azure DevOps
devops-mckendrick / DevOpsContainerBuild / Pipelines
Search
DevOpsContainerBuild
Overview
Pipelines
Pipelines
Environments
Releases
Library
Task groups
Deployment groups
Project settings
Create your first Pipeline
Automate your build and release processes using our wizard, and go from code to cloud-hosted within minutes.
Create Pipeline

Figure 15.26: Viewing the Pipelines page for the first time

As you may have guessed, you need to click on **Create Pipeline**, this will ask you for several pieces of information:

1. **Where is your code?** Select GitHub. You will notice that YAML is next to it. We will be talking about the YAML file once we have the pipeline configured.

2. Follow the onscreen instructions to link Azure DevOps to your GitHub account. Once it is linked, you will be asked to **Select a repository**. Select the repository that you forked earlier.
3. If the `azure-pipelines.yml` file is not automatically selected and you stay on the **Configure** screen, click on the **Existing Azure Pipelines YAML file** option, select the file from the drop-down list, and then click on **Continue**.
4. The **Review** page gives you the option to **Review your pipeline YAML** file, as well as the option to **Run** it; however, before we do, click on **Variables**.
5. We need to add two variables. The first will let the pipeline know the name of the service connection to Docker Hub that we configured earlier in this section and the second one will let the pipeline know the name of the Docker Hub repository we would like our Azure DevOps pipeline to push the image to once it has been built.
6. Click on the **New variable** button, enter `targetRegistry` in the **Name**, and then for the **Value**, enter whatever you called the service connection, which in my case was `Docker`. Click on the **OK** button and then, once you are back on the variables page, click on **+**. For the second variable, give it a **Name** of `targetRepo`, then for the **Value**, enter your Docker Hub username and then the repository name—for example, I entered `russmckendrick/AzureDevOpsBuild`. Click on **OK** and then **Save**. Once saved, click on the **Run** button to trigger your build.

The `azure-pipeline.yml` file looks like the following. First, we have the **trigger** configuration; this is set to **master**, which means that a build is triggered every time that the master branch is updated:

```
trigger:
- master
```

Next up, we have **pool**. This tells Azure DevOps which virtual image to launch when the pipeline is being executed; as you can see, we are using Ubuntu:

```
pool:
  vmImage: 'ubuntu-latest'
```

The remainder of the `azure-pipeline.yml` file is the build **steps**; these are the tasks that will build and push our container. They are executed in the order in which they are defined. Our first step is to use the `Docker@2` task to log in to Docker Hub:

```
steps:
- task: 'Docker@2'
```

```
  displayName: 'Login to Docker Hub'
  inputs:
    command: 'login'
    containerRegistry: '$(targetRegistry)'
```

We are using the variable that we defined when setting up the pipeline by entering `$(targetRegistry)`. This lets the task know which service connection to use. The next task builds and pushes our container image:

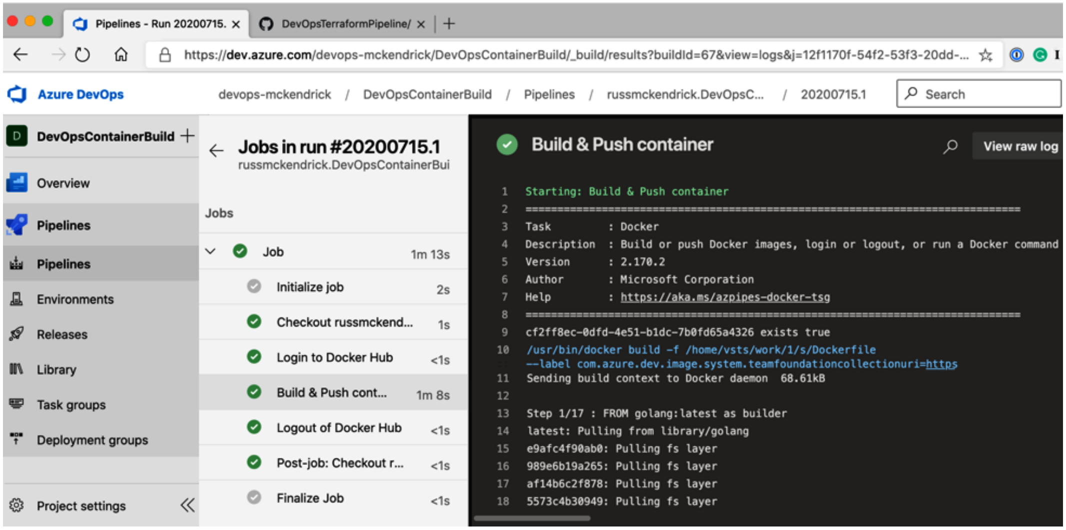
```
- task: Docker@2
  displayName: 'Build & Push container'
  inputs:
    command: 'buildAndPush'
    containerRegistry: '$(targetRegistry)'
    repository: '$(targetRepo)'
    tags: |
      latest
```

As you can see, the syntax is easy to follow. We are also using the second variable `$(targetRepo)` to define the target for our image to be pushed to. The final task logs out of Docker Hub:

```
- task: 'Docker@2'
  displayName: 'Logout of Docker Hub'
  inputs:
    command: 'logout'
    containerRegistry: '$(targetRegistry)'
```

While the last task is probably not really needed, as, while Azure DevOps spins up to build our image, the virtual machine is terminated once the build finishes, it will also be terminated if there are any errors, so we do not have to worry about the virtual machine being reused or our login being accessed by a third party.

A completed pipeline run looks something like the following:


Pipelines - Run 20200715.
DevOpsTerraformPipeline/
https://dev.azure.com/devops-mckendrick/DevOpsContainerBuild/_build/results?buildId=67&view=logs&j=12f1170f-54f2-53f3-20dd-...
Azure DevOps
devops-mckendrick / DevOpsContainerBuild / Pipelines / russmckendrick.DevOpsC... / 20200715.1
Search
DevOpsContainerBuild
Overview
Pipelines
Pipelines
Environments
Releases
Library
Task groups
Deployment groups
Project settings
Jobs in run #20200715.1
russmckendrick.DevOpsContainerBui
Jobs
View raw log

Job	1m 13s
Initialize job	2s
Checkout russmckend...	1s
Login to Docker Hub	<1s
Build & Push cont...	1m 8s
Logout of Docker Hub	<1s
Post-job: Checkout r...	<1s
Finalize Job	<1s

Build & Push container

```
Starting: Build & Push container
==============================================================================
Task         : Docker
Description  : Build or push Docker images, login or logout, or run a Docker command
Version      : 2.170.2
Author       : Microsoft Corporation
Help         : https://aka.ms/azpipes-docker-tsg
==============================================================================
cf2ff8ec-0dfd-4e51-b1dc-7b0fd65a4326 exists true
/usr/bin/docker build -f /home/vsts/work/1/s/Dockerfile
--label com.azure.dev.image.system.teamfoundationcollectionuri=https
Sending build context to Docker daemon  68.61kB

Step 1/17 : FROM golang:latest as builder
latest: Pulling from library/golang
e9afc4f90ab0: Pulling fs layer
989e6b19a265: Pulling fs layer
af14b6c2f878: Pulling fs layer
5573c4b30949: Pulling fs layer
```

Figure 15.27: A completed pipeline run

Once finished, you should see the newly built container in your Docker Hub account. As mentioned before we started to configure our Azure DevOps pipeline, we have hardly scratched the surface of what Azure DevOps can do; see the *Further reading* section of this chapter for some interesting links on Azure DevOps.

Next, we are going to take a look at how to monitor our containers and Docker hosts.

Monitoring Docker and Kubernetes

In *Chapter 4*, *Managing Containers*, we discussed the `docker container top` and `docker container stats` commands. You may recall that both of these commands show real-time information only—there is no historical data that is kept.

This is great if you are trying to debug a problem as it is running or want to quickly get an idea of what is going on inside your containers; however, it is not too helpful if you need to look back at a problem. For example, you may have configured your containers to restart if they have become unresponsive. While that will help with the availability of your application, it isn't much of a help if you need to look at why your container became unresponsive.

In the GitHub repository in the `/chapter14` folder, there is a folder called `prometheus` in which there is a Docker Compose file that launches three different containers on two networks. Rather than looking at the Docker Compose file itself, let's take a look at a visualization:

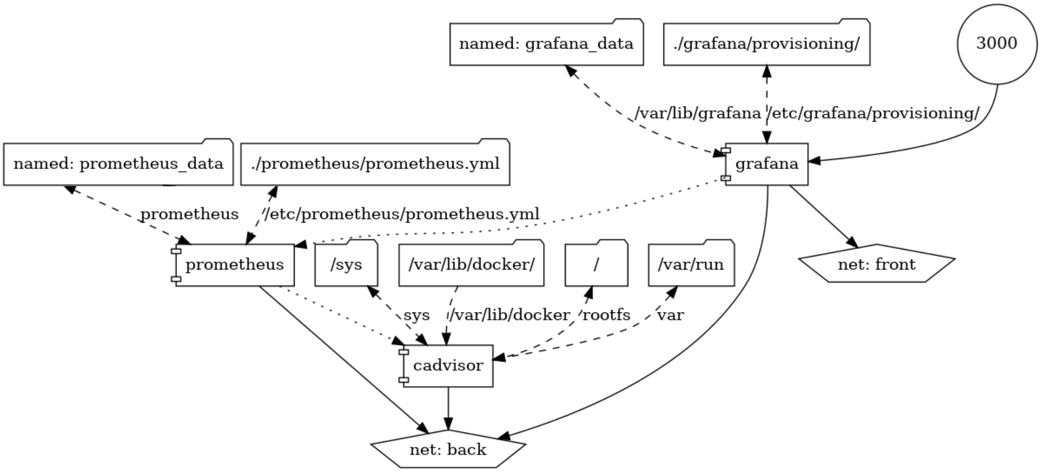


Figure 15.28: Visualization of the Prometheus Docker Compose file

You can generate this yourself by running the following command:

```
$ docker container run --rm -it --name dcv -v $(pwd):/input
pmsipilot/docker-compose-viz render -m image docker-compose.yml
```

As you can see, there is a lot going on. The three services that we are running are as follows:

- Cadvisor
- Prometheus
- Grafana

Before we launch and configure our Docker Compose services, we should talk about why each one is needed, starting with `cadvisor`:

The `cadvisor` service is a project that was released by Google. As you can see from the Docker Hub username in the image we are using, the service section in the Docker Compose file looks like the following:

```
  cadvisor:
    image: google/cadvisor:latest
    container_name: cadvisor
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    restart: unless-stopped
    expose:
      - 8080
    networks:
      - back
```

We are mounting the various parts of our host's filesystem to allow `cadvisor` access to our Docker installation in much the same way as we did in *Chapter 9*, *Portainer – A GUI for Docker*. The reason for this is that, in our case, we are going to be using `cadvisor` to collect statistics on our containers. While it can be used as a standalone container-monitoring service, we do not want to publicly expose the `cadvisor` container. Instead, we are just making it available to other containers within our Docker Compose stack on the back network.

The `cadvisor` service is a self-contained web frontend to the Docker container stat command, displaying graphs and allowing you to drill down from your Docker host into your containers using an easy-to-use interface; however, it doesn't keep more than five minutes' worth of metrics.

As we are attempting to record metrics that can be available hours or even days later, having no more than five minutes' worth of metrics means that we are going to have to use additional tools to record the metrics it processes. The `cadvisor` service exposes the information that we want to record in our containers as structured data at `http://cadvisor:8080/metrics/`.

We will look at why this is important in a moment. The `cadvisor` endpoint is being scraped automatically by our next service, `prometheus`. This is where most of the heavy lifting happens. The `prometheus` is a monitoring tool that is written and open sourced by SoundCloud:

```
  prometheus:
    image: prom/prometheus
    container_name: prometheus
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.
```

```
yml
      - prometheus_data:/prometheus
    restart: unless-stopped
    expose:
      - 9090
    depends_on:
      - cadvisor
    networks:
      - back
```

As you can see from the preceding service definition, we are mounting a configuration file called `./prometheus/prometheus.yml` and a volume called `prometheus_data`. The configuration file contains information about the sources we want to scrape, as you can see from the following configuration:

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s
  external_labels:
      monitor: 'monitoring'
rule_files:

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']
```

We are instructing Prometheus to scrape data from our endpoints every 15 seconds. The endpoints are defined in the `scrape_configs` section, and as you can see, we have `cadvisor` defined in there, as well as Prometheus itself. The reason we are creating and mounting the `prometheus_data` volume is that Prometheus is going to be storing all of our metrics, so we need to keep it safe.

At its core, Prometheus is a time-series database. It takes the data it has scraped, processes it to find the metric name and value, and then stores it along with a timestamp.

Prometheus also comes with a powerful query engine and API, making it the perfect database for this kind of data. While it does come with basic graphing capabilities, it is recommended that you use *Grafana*, which is our final service, and also the only one to be exposed publicly.

Grafana is an open source tool for displaying monitoring graphs and metric analytics, which allows you to create dashboards using time-series databases, such as Graphite, InfluxDB, and also Prometheus. There are also further backend database options that are available as plugins.

The Docker Compose definition for Grafana follows a similar pattern to our other services:

```
  grafana:
    image: grafana/grafana
    container_name: grafana
    volumes:
      - grafana_data:/var/lib/grafana
      - ./grafana/provisioning/:/etc/grafana/provisioning/
    env_file:
      - ./grafana/grafana.config
    restart: unless-stopped
    ports:
      - 3000:3000
    depends_on:
      - prometheus
    networks:
      - front
      - back
```

We are using the `grafana_data` volume to store Grafana's own internal configuration database, and rather than storing the environment variables in the Docker Compose file, we are loading them from an external file called `./grafana/grafana.config`.

The variables are as follows:

```
GF_SECURITY_ADMIN_USER=admin
GF_SECURITY_ADMIN_PASSWORD=password
GF_USERS_ALLOW_SIGN_UP=false
```

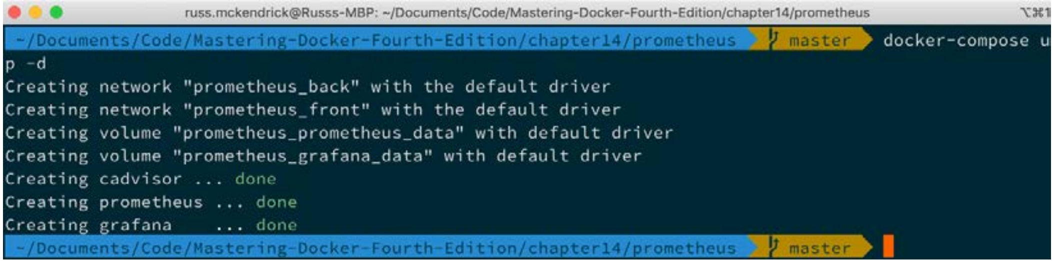
As you can see, we are setting the username and password here, so having them in an external file means that you can change these values without editing the core Docker Compose file.

Now that we know the role that each of the three services fulfills, let's launch them.

To do this, simply run the following commands from the `prometheus` folder:

```
$ docker-compose pull
$ docker-compose up -d
```

This will create a network and the volumes and pull the images from the Docker Hub. It will then go about launching the three services:



russ.mckendrick@Russs-MBP: ~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/prometheus
⌥⌘1

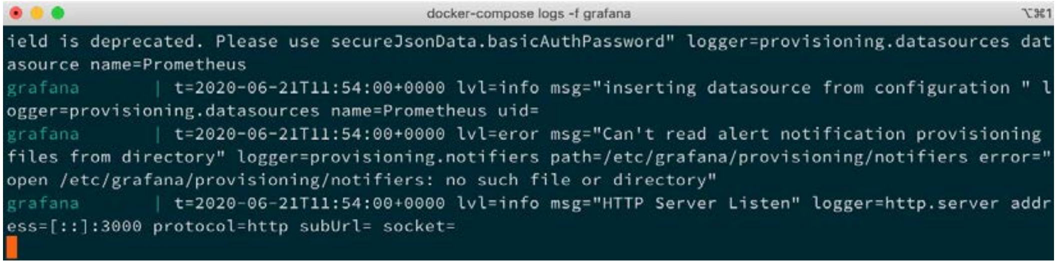
```
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/prometheus  master  docker-compose u
p -d
Creating network "prometheus_back" with the default driver
Creating network "prometheus_front" with the default driver
Creating volume "prometheus_prometheus_data" with default driver
Creating volume "prometheus_grafana_data" with default driver
Creating cadvisor ... done
Creating prometheus ... done
Creating grafana    ... done
~/Documents/Code/Mastering-Docker-Fourth-Edition/chapter14/prometheus  master
```

Figure 15.29: Running docker-compose up -d to launch our Prometheus application

You may be tempted to go immediately to your Grafana dashboard. If you did, you would not see anything, as Grafana takes a few minutes to initialize itself. You can follow its progress by looking at the logs:

```
$ docker-compose logs -f grafana
```

The output of the command is given here:



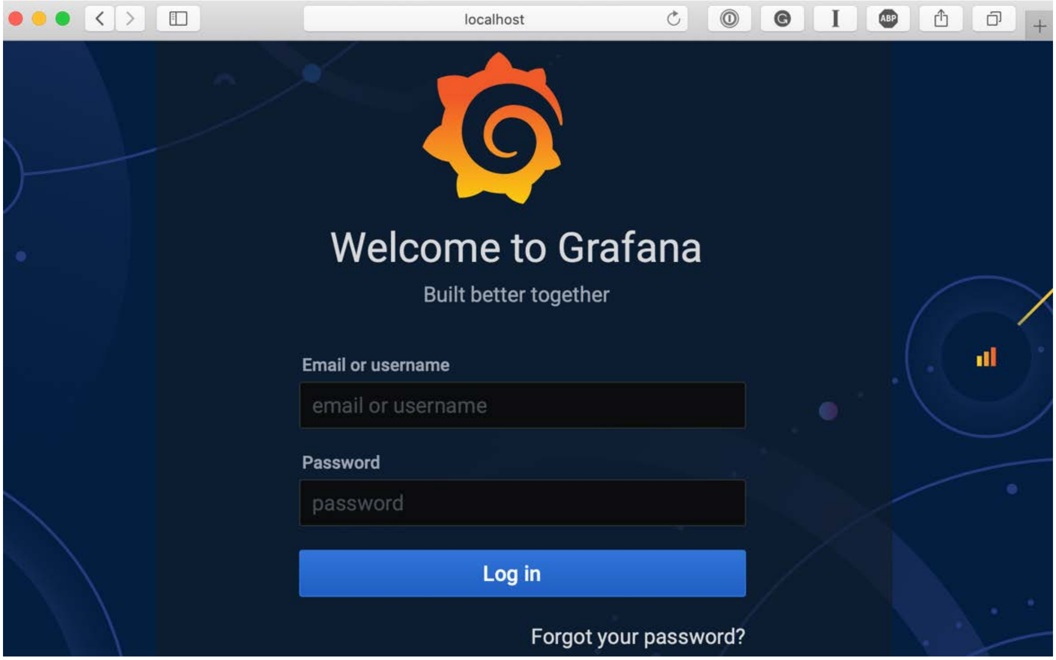
```
ield is deprecated. Please use secureJsonData.basicAuthPassword" logger=provisioning.datasources dat
asource name=Prometheus
grafana        | t=2020-06-21T11:54:00+0000 lvl=info msg="inserting datasource from configuration " l
ogger=provisioning.datasources name=Prometheus uid=
grafana        | t=2020-06-21T11:54:00+0000 lvl=eror msg="Can't read alert notification provisioning
files from directory" logger=provisioning.notifiers path=/etc/grafana/provisioning/notifiers error="
open /etc/grafana/provisioning/notifiers: no such file or directory"
grafana        | t=2020-06-21T11:54:00+0000 lvl=info msg="HTTP Server Listen" logger=http.server addr
ess=[::]:3000 protocol=http subUrl= socket=
```

Figure 15.30: Checking the logs to see if Grafana is ready

Once you see the `HTTP Server Listen` message, Grafana will be available. From Grafana version `5`, you can import data sources and dashboards, which is why we are mounting the `./grafana/provisioning/` folder from our host machine to `/etc/grafana/provisioning/`.

This folder contains the configuration that automatically configures Grafana to talk to our Prometheus service and imports the dashboard, which will display the data that Prometheus is scraping from `cadvisor`.

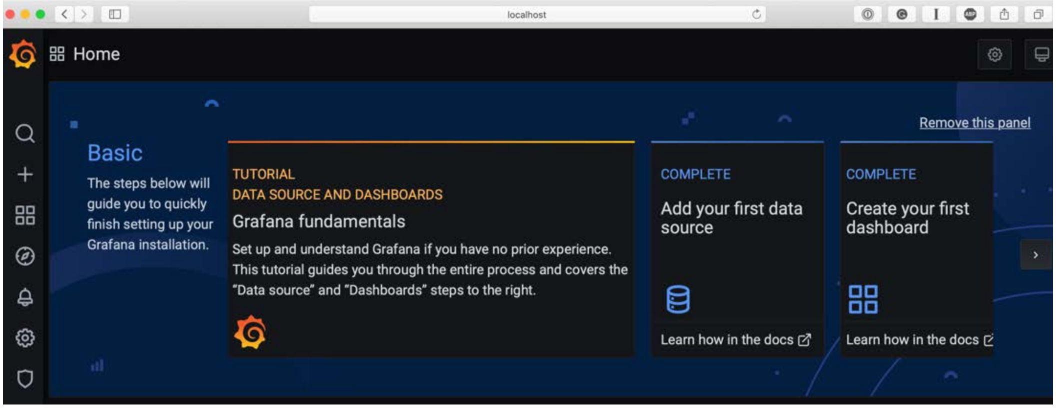
Open your browser and enter `http://localhost:3000/`; you should be greeted with a login screen:


Welcome to Grafana

Built better together

Figure 15.31: The Grafana login page

Enter the **Username** as `admin` with a **Password** of `password`. Once logged in, if you have configured the data source, you should see the following page:



Basic

The steps below will guide you to quickly finish setting up your Grafana installation.

TUTORIAL
DATA SOURCE AND DASHBOARDS

Grafana fundamentals

Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.

COMPLETE

Add your first data source

Learn how in the docs

COMPLETE

Create your first dashboard

Learn how in the docs

Figure 15.32: Logging into Grafana

As you can see, the initial steps of **Add your first data source** and **Create your first dashboard** have all been completed. Clicking on the **Home** button in the top left will bring up a menu that lists the available dashboards:

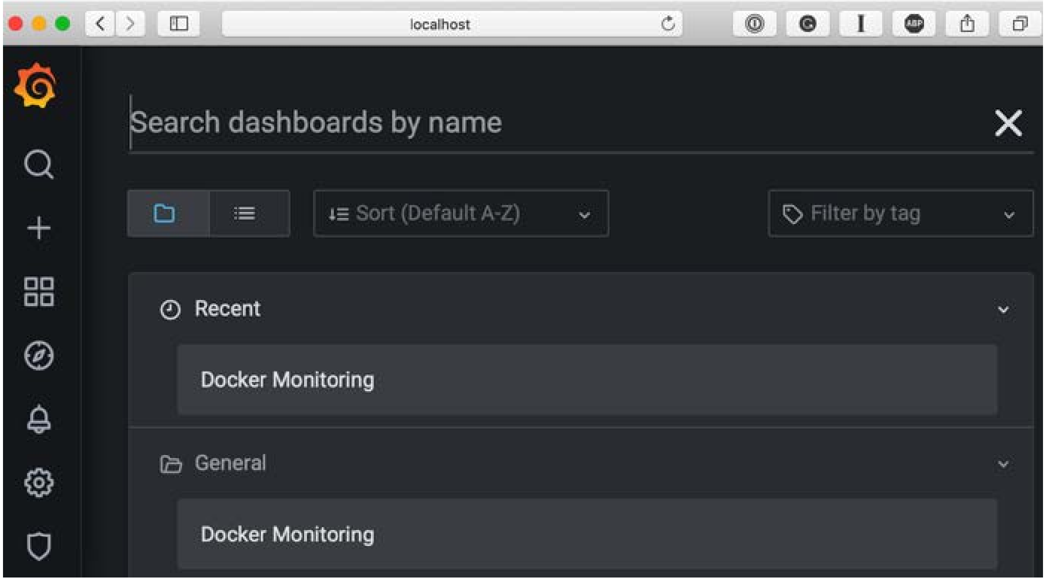


Figure 15.33: Viewing the available dashboards

As you can see, we have one called **Docker Monitoring**. Clicking on it will take you to the following page:

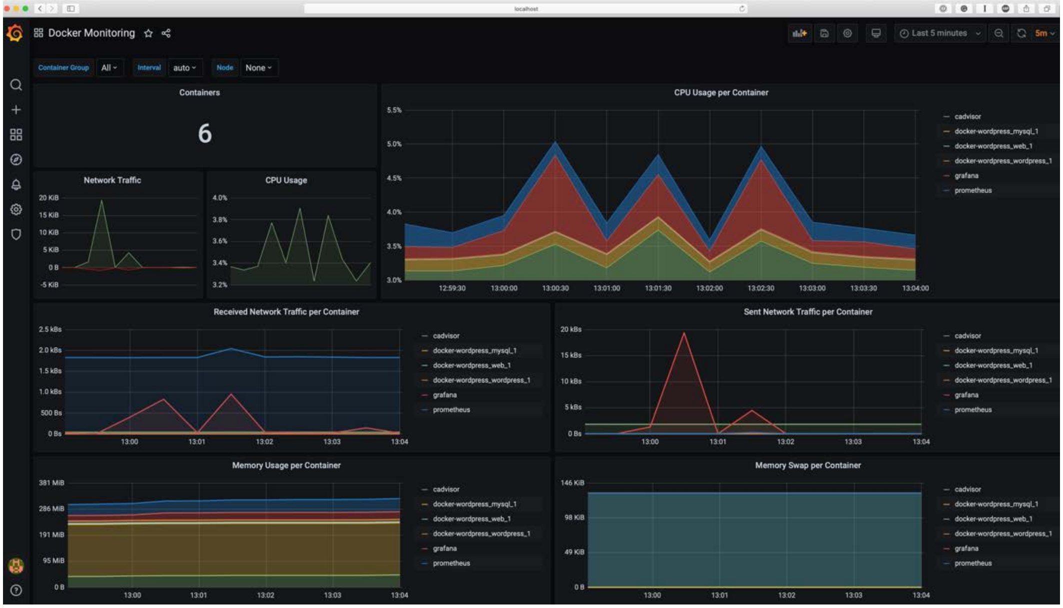


Figure 15.34: The Docker Monitoring dashboard

As you can see from the timing information in the top right of the screen, by default, it displays the last five minutes' worth of data. Clicking on it will allow you to change the time frame displays. For example, the following screen shows the last 15 minutes, which is obviously more than the five minutes that `cadvisor` is recording:

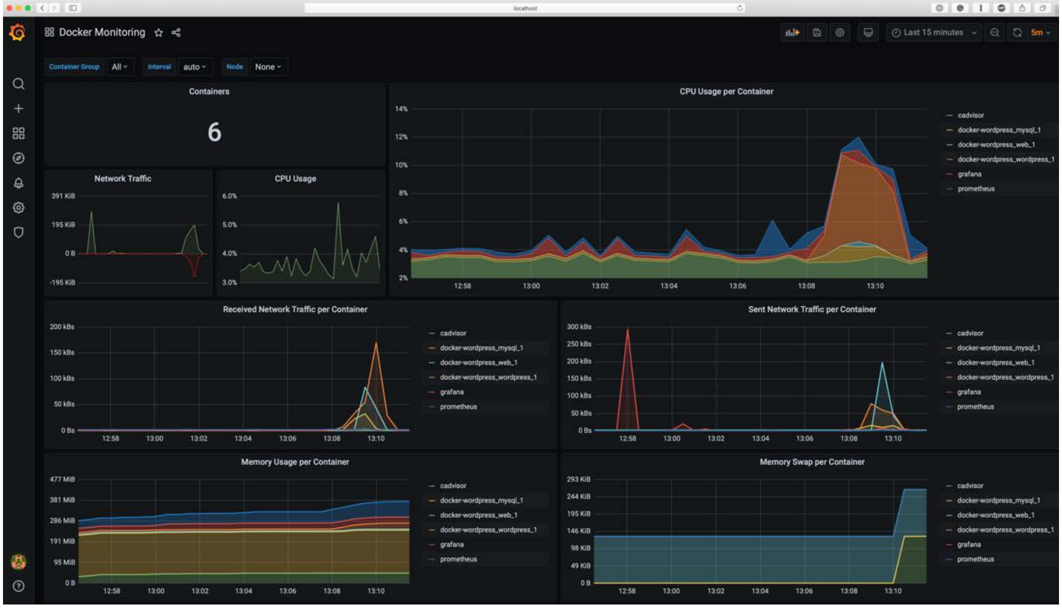


Figure 15.35: Viewing 15 minutes of data

I have already mentioned that this is a complex solution; eventually, Docker will expand the recently built-in Prometheus endpoint, which currently only exposes information about Docker Engine and not the containers themselves. For more information on the built-in endpoint, check out the official Docker documentation, which can be found at `https://docs.docker.com/config/daemon/prometheus/`.

There are other monitoring solutions out there; most of them take the form of third-party **software as a service** (**SaaS**). As you can see from the list of services in the *Further reading* section, there are a few well-established monitoring solutions out there. In fact, you may already be using them, so it would be easy for you when expanding your configuration to take this into account when monitoring your containers.

What about Kubernetes? you may be asking yourself. I have already mentioned that Prometheus was originally developed by *SoundCloud*, but it was also one of the first projects outside of Kubernetes to be donated to the **Cloud Native Computing Foundation** (**CNCF**).

This means that there is support for Prometheus within Kubernetes and external services, such as Azure AKS—for example, Azure Monitor has seamless integration with Prometheus.

For a demonstration of this, see the Azure Friday presentation *How to use Prometheus to monitor containers in Azure Monitor* by *Keiko Harada with Scott Hanselman*, which can be found at the Microsoft Azure YouTube channel at `https://www.youtube.com/watch?v=5ARJ6DzqTYE`.

What does production look like?

For the final section of this chapter, we are going to discuss what production should look like. This section isn't going to be as long as you think it will be, as the sheer number of options that are available means that it would be impossible to cover them all. You should also already have a good idea of what would work best for you based on the previous sections and chapters.

Instead, we are going to be looking at some questions that you should be asking yourself when planning your environments.

Your Docker hosts

Docker hosts are the key component of your environment. Without these, you won't have anywhere to run your containers. As we have already seen in previous chapters, there are a few considerations to bear in mind when it comes to running your Docker hosts.

The first thing you need to take into account is that, if your hosts are running Docker, they should not run any other services.

Mixing of processes

You should resist the temptation of quickly installing Docker on an existing host and launching a container. This might not only have implications for security, with you having a mixture of isolated and nonisolated processes on a single host, but it can also cause performance issues as you are not able to add resource limits to your noncontainerized applications, meaning that, potentially, they can also have a negative impact on your running containers.

Multiple isolated Docker hosts

If you have more than a few Docker hosts, how are you going to manage them? Running a tool, such as Portainer, is great, but it can become troublesome when attempting to manage more than a few hosts. Also, if you are running multiple isolated Docker hosts, you do not have the option of moving containers between hosts.

Sure, you can use tools such as Weave Net to span the container network across multiple individual Docker hosts. Depending on your hosting environment, you may also have the option of creating volumes on external storage and presenting them to Docker hosts as needed, but you are very much creating a manual process to manage the migration of containers between hosts.

Routing to your containers

You need to consider how you are going to route requests among your containers if you have multiple hosts.

For example, if you have an external load balancer, such as an ELB in AWS, or a dedicated device in front of an on-premise cluster, do you have the ability to dynamically add routes for traffic hitting port **x** on your load balancer to port **y** on your Docker hosts, at which point the traffic is then routed through to your container?

If you have multiple containers that all need to be accessible on the same external port, how are you going handle that?

Do you need to install a proxy, such as Traefik, HAProxy, or NGINX to accept and then route your requests based on virtual hosts based on domains or subdomains, rather than just using port-based routing?

Clustering

A lot of what we have discussed in the previous section can be solved by introducing clustering tools, such as Docker Swarm and Kubernetes.Let's quickly discuss some of the things that you should be asking yourself when assessing clustering technologies.

Compatibility

Even though an application might work fine on a developer's local Docker installation, you need to be able to guarantee that if you take the application and deploy it to, for example, a Kubernetes cluster, it works in the same way.

Nine times out of ten, you will not have a problem, but you do need to consider how the application is communicating internally with other containers within the same application set.

Reference architectures

Are there reference architectures available for your chosen clustering technology? It is always best to check when deploying a cluster. There are best-practice guides that are close to or match your proposed environment. After all, no one wants to create one big single point of failure.

Also, what are the recommended resources? There is no point in deploying a cluster with five management nodes and a single Docker host, just like there is little point in deploying five Docker hosts and a single management server, as you have quite a large single point of failure.

What supporting technologies does your cluster technology support (for example, remote storage, load balancers, and firewalls)?

Cluster communication

What are the requirements when it comes to the cluster communicating with either management or Docker hosts? Do you need an internal or separate network to isolate the cluster traffic?

Can you easily lock a cluster member down to only your cluster? Is the cluster communication encrypted? What information about your cluster could be exposed? Does this make it a target for hackers?

What external access does the cluster need to APIs, such as your public cloud providers? How securely are any API/access credentials stored?

Image registries

How is your application packaged? Have you baked the code into the image? If so, do you need to host a private local image registry or are you okay with using an external service, such as Docker Hub, **Docker Trusted Registry** (**DTR**), or Quay?

If you need to host your own private registry, where in your environment should it sit? Who has or needs access? Can it hook into your directory provider, such as an Active Directory installation?

Summary

In this chapter, we looked at a few different workflows for Docker, along with how to get some monitoring for your containers and Docker hosts up and running.

The best thing you can do when it comes to your own environment is build a proof of concept and try as hard as you can to cover every disaster scenario you can think of. You can get a head start by using the container services provided by your cloud provider or by looking for a good reference architecture, which should both reduce your trial and error rates.

In the next chapter, we are going to take a look at what your next step in the world of containers could be.

Questions

1. Which container serves our WordPress website?
2. Why doesn't the `wp` container remain running?
3. In minutes, how long does `cadvisor` keep metrics for?
4. What Docker Compose command can be used to remove everything to do with the application?

Further reading

You can find details on the software we have used in this chapter at the following sites:

- WordPress: `http://wordpress.org/`
- WP-CLI: `https://wp-cli.org/`
- PHP-FPM: `https://php-fpm.org/`
- Cadvisor: `https://github.com/google/cadvisor/`
- Prometheus: `https://prometheus.io/`
- Grafana: `https://grafana.com/`

- Prometheus data model: `https://prometheus.io/docs/concepts/data_model/`
- Traefik: `https://containo.us/traefik/`
- HAProxy: `https://www.haproxy.org/`
- NGINX: `https://nginx.org/`

For more information on Docker and Azure DevOps, go to the following links:

- Azure DevOps Docker Build Task: `https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/build/docker?view=azure-devops`
- Azure DevOps Docker Compose Build Task: `https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/build/docker-compose?view=azure-devops`
- Azure DevOps and Azure Container Registry: `https://docs.microsoft.com/en-us/azure/devops/pipelines/ecosystems/containers/acr-template?view=azure-devops`
- Azure DevOps titles at Packt Publishing: `https://www.packtpub.com/catalogsearch/result/?q=Azure%20DevOps`

Other externally hosted Docker monitoring platforms include the following:

- Sysdig Cloud: `https://sysdig.com/`
- Datadog: `https://docs.datadoghq.com/agent/docker/?tab=standard`
- SignalFx: `https://signalfx.com/docker-monitoring/`
- New Relic: `https://newrelic.com/partner/docker`
- Sematext: `https://sematext.com/docker/`

There are also other self-hosted options, such as the following:

- Elastic Beats: `https://www.elastic.co/products/beats`
- Sysdig: `https://sysdig.com/opensource/`
- Zabbix: `https://github.com/monitoringartist/zabbix-docker-monitoring`

The following list shows some extensions for other IDEs:

- Atom Docker Package: `https://atom.io/packages/docker`
- Sublime Text Docker Plugin: `https://github.com/domeide/sublime-docker`
- Jetbrains Docker support: `https://www.jetbrains.com/help/idea/docker.html`

16
Next Steps with Docker

You've made it to the last chapter of this book, and you've stuck with it until the end! In this chapter, we will look at the Moby Project and how you can contribute to Docker, as well as to the community. We will then finish this chapter with a quick overview of the Cloud Native Computing Foundation.

The following topics will be covered in this chapter:

- The Moby Project
- Contributing to Docker
- The Cloud Native Computing Foundation

The Moby Project

One of the announcements made at DockerCon 2017 was the Moby Project. When this project was announced, I had a few questions about what the project was from work colleagues, because on the face of it, Docker had appeared to have released another container system.

I answered with the following:

> *The Moby Project is the open source framework created by Docker (the company) that allows it, and anyone else who wishes to contribute to the project, to assemble container systems "without having to reinvent the wheel".*
>
> *Think of the framework as a set of building blocks made up of dozens of components that allow you assemble your own custom container platform. The Moby project is used to create the open source community edition of Docker and the commercially supported Docker Enterprise edition.*

For anyone who asks for an example of a similar project that combines a bleeding-edge version, a stable open source release, and an enterprise supported version, I explain what Red Hat does with Red Hat Enterprise Linux:

> *Think of it like the approach Red Hat have taken with Red Hat Enterprise Linux. You have Fedora, which is the bleeding-edge version development playground for Red Hat's operating system developers to introduce new packages and features, and also to remove old, outdated components.*
>
> *Typically, Fedora is a year or two ahead of the features found in Red Hat Enterprise Linux, which is the commercially supported long-term release based on the work done in the Fedora project; as well as this release, you also have the community support version in the form of CentOS.*

You may be thinking to yourself, *why has this only been mentioned right at the very end of this book?* The project isn't really designed for end users. It is intended for use by software engineers, integrators, and enthusiasts looking to experiment with, invent, and build systems based on containers – which is not necessarily Docker.

At the time of writing, due changes at Docker (the company) with the Docker Enterprise sale to Mirantis and Docker refocusing their attention back onto developers, there are talks of moving the Moby code back to Docker, so I will not go into any more detail about the Moby Project as it is likely to change by the time you read this; instead, I would recommend bookmarking the following pages to keep up to date with this how the project develops:

- The project's main website, at `https://mobyproject.org/`
- Moby Project GitHub pages, at `https://github.com/moby/`

- The Moby Project Twitter account, a good source of news and links to how-tos, at `https://twitter.com/moby/`
- Discussion around moving Moby back to Docker: `https://github.com/moby/moby/issues/40222` and `https://www.theregister.com/2019/11/22/moby_docker_naming/`

Contributing to Docker

So, you want to help contribute to Docker? Do you have a great idea that you would like to see in Docker or one of its components? Let's get you the information and tools that you need to do that. If you aren't a programmer-type person, there are other ways you can help contribute as well. Docker has a massive audience, and another way you can help contribute is to help with supporting other users with their services.

Let's learn how you can do that as well.

Contributing to the code

One of the biggest ways you can contribute to Docker is by helping with the Moby code, as this is the upstream project that Docker is based on.

Since Moby is open source, you can download the code to your local machine and work on new features and present them as pull requests back to Moby. They will then get reviewed on a regular basis, and if they feel what you have contributed should be in the service, they will approve the pull request. This can be very humbling when it comes to knowing that something you have written has been accepted.

You first need to know how you can get set up to contribute: this is pretty much everything for Docker (`https://github.com/docker/`) and Moby Project (`https://github.com/moby/`).

But how do we go about getting set up to help contribute? The best place to start is by following the guide that can be found on the official Moby documentation at `https://github.com/moby/moby/blob/master/CONTRIBUTING.md`.

As you may have already guessed, you do not need much to get a development environment up and running as a lot of development is done within containers. For example, other than having a GitHub account, Moby lists the following three pieces of software as the bare minimum:

- Git: `https://git-scm.com/`
- Make: `https://www.gnu.org/software/make/`
- Docker: If you made it this far, you shouldn't need a link

You can find more details on how to prepare your own Docker development for Mac and Linux at `https://github.com/moby/moby/blob/master/docs/contributing/software-required.md` and for Windows at `https://github.com/moby/moby/blob/master/docs/contributing/software-req-win.md`.

To be a successful open source project, there have to be some community guidelines. I recommend reading through the excellent quick start guide that can be found at `https://github.com/moby/moby/blob/master/CONTRIBUTING.md#moby-community-guidelines` as well as the more detailed contribution workflow documentation at `https://github.com/moby/moby/blob/master/docs/contributing/who-written-for.md`.

Offering Docker support

You can also contribute to Docker by other means beyond contributing to the Docker code or feature sets. You can help by using the knowledge you have obtained to help others in their support channels. The community is very open, and someone is always willing to help.

I find it of great help when I run into something and I am found scratching my head. It's also nice to get help but to also contribute to others; this is a nice give and take. It also is a great place to harvest ideas for you to use. You can see what questions others are asking based on their setups and it could spur ideas that you may want to think about using in your environment.

You can also follow the GitHub issues that are brought up regarding the services. These could be feature requests and how Docker may implement them, or they could be issues that have cropped up through the use of services. You can help test out the issues that others are experiencing to see whether you can replicate the issue or whether you find a possible solution to their issue.

Docker has a very active community that can be found at `https://www.docker.com/docker-community`; here, you will not only be able to see the latest community news and events, but you will also be able to chat with Docker users and developers in their Slack channels. At the time of writing this book, there are over 80 channels covering all sorts of topics, such as Docker for Mac, Docker for Windows, Alpine Linux, Swarm, Storage, and Network to name but a few, with hundreds of active users at any one time.

Finally, there are also the Docker forums, which can be found at `https://forums.docker.com/`. These are a good source if you want to search for topics/problems or keywords.

The Docker Community is governed by a code of conduct that covers both how their staff and community as a whole should act. It is open source and licensed under the Creative Commons Attribution 3.0, and states the following:

> *We are dedicated to providing a harassment-free experience for everyone, and we do not tolerate harassment of participants in any form. We ask you to be considerate of others and behave professionally and respectfully to all other participants. This code and related procedures also apply to unacceptable behavior occurring outside the scope of community activities, in all community venues— online and in-person— as well as in all one-on-one communications, and anywhere such behavior has the potential to adversely affect the safety and well-being of community members. Exhibitors, speakers, sponsors, staff and all other attendees at events organized by Docker, Inc (DockerCon, meetups, user groups) or held at Docker, Inc facilities are subject to these Community Guidelines and Code of Conduct.*
>
> *Diversity and inclusion make the Docker community strong. We encourage participation from the most varied and diverse backgrounds possible and want to be very clear about where we stand.*
>
> *Our goal is to maintain a safe, helpful, and friendly Docker community for everyone, regardless of experience, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, nationality, or other protected categories under applicable law.*

The full code of conduct can be found at `https://github.com/docker/code-of-conduct/`.

Other contributions

There are other ways to contribute to Docker as well. You can do things such as promoting the service and gathering interest at your institution. You can start this communication through your own organization's means of communications, whether that be email distribution lists, group discussions, IT roundtables, or regularly scheduled meetings.

You can also schedule meetups within your organization to get people talking. These meetups are designed to not only include your organization, but the city or town members that your organization is in, in order to get more widespread communication and promotion of the services.

You can search whether there are already meetups in your area by visiting `https://www.docker.com/community/meetup-groups/`.

The Cloud Native Computing Foundation

We discussed The Cloud Native Computing Foundation briefly in *Chapter 11*, *Docker and Kubernetes*. The Cloud Native Computing Foundation, or CNCF for short, was founded to provide a vendor-neutral home for projects that allow you to manage your containers and microservices architectures.

Its membership includes Docker, Amazon Web Services, Google Cloud, Microsoft Azure, Red Hat, Oracle, VMWare, and Digital Ocean, to name a few. In June 2020, the Linux Foundation reported that CNCF had 452 members. These members not only contribute projects but also engineering time, code, and resources.

Graduated projects

At the time of writing this book, there are ten graduated projects, some of which we have discussed in previous chapters. The two which have already covered are also the most well-known out of the ten projects that are maintained by the foundation are as follows:

- **Kubernetes** (`https://kubernetes.io`): This was the first project to be donated to the Foundation. As we have already mentioned, it was originally developed by Google and now counts more than 2,300 contributors across members of the foundation as well as the open source community.
- **Prometheus** (`https://prometheus.io`): This project was donated to the foundation by SoundCloud. As we saw in *Chapter 15*, *Docker Workflows*, it is a real-time monitoring and alerting system that's backed by a powerful time-series database engine.

There is also the following:

- **Envoy** (`https://www.envoyproxy.io/`): Originally created inside Lyft and used companies such as Apple, Netflix, and Google, Envoy is a highly optimized service mesh that provides load balancing, tracing, and observability of the database and network activity across your environment.
- **CoreDNS** (`https://coredns.io/`): This is a small, flexible, extendable, and highly optimized DNS server that's written in Go and designed from the ground up to run in an infrastructure that can be running thousands of containers.
- **Containerd** (`https://containerd.io/`): We briefly mentioned Containerd in *Chapter 1*, *Docker Overview*, as being one of the open source projects that Docker has been working on, and we also used it in *Chapter 12*, *Discovering more Kubernetes options*. It is a standard container runtime that allows developers to embed a runtime that can manage both Docker- and also OCI-compliant images in their platforms or applications.
- **Fluentd** (`https://www.fluentd.org/`): This tool allows you to collect log data from a large number of sources and then route the logging data to a number of log management, database, archiving, and alerting systems such as Elastic Search, AWS S3, MySQL, SQL Server, Hadoop, Zabbix, and DataDog, to name a few.
- **Jaeger** (`https://www.jaegertracing.io/`): This is a fully distributed tracing system that was originally developed by Uber to monitor its extensive microservices environment. Now in use by companies such as Red Hat, it features a modern UI and native support for OpenTracing and various backend storage engines. It has been designed to integrate with other CNCF projects such as Kubernetes and Prometheus.
- **Vitess** (`https://vitess.io/`): This has been a core component of the MySQL database infrastructure of YouTube since 2011. It is a clustering system that horizontally scales MySQL via sharding.
- **Helm** (`https://helm.sh/`): Built for Kubernetes, Helm is a package manager that allows users to package their Kubernetes applications in an easily distributable format, and has quickly become a standard.

To graduate, a project must have done the following:

- Adopted the CNCF code of conduct, which is similar to the one published by Docker. The full code of conduct can be found at `https://github.com/cncf/foundation/blob/master/code-of-conduct.md`.
- Obtained a **Linux Foundation** (**LF**) **Core Infrastructure Initiative** (**CII**) Best Practices badge, which demonstrates that the project is being developed using an established set of best practices – the full criteria of which can be found at `https://github.com/coreinfrastructure/best-practices-badge/blob/master/doc/criteria.md.`
- Acquired at least two organizations with committers to the project.
- Defined the committer process and project governance publicly via `GOVERNANCE.md` and `OWNERS.md` files in the project's repo.
- Publicly listed the projects adopters in an `ADOPTERS.md` file or by logos on the project's website.
- Received a super majority vote from the **Technical Oversight Committee** (**TOC**). You can find out more about the committee at `https://github.com/cncf/toc`.

There is also another project status, which is where the majority of projects currently are.

Incubating projects

Projects at the incubating stage should eventually have a graduated status. The following projects have all done the following:

- Demonstrated that the project is in use by a minimum of three independent end users (not the originator of the project)
- Gained a healthy number of contributors, both internally and externally
- Demonstrated growth and a good level of maturity

The TOC is heavily involved in working with projects to ensure that the levels of activity are enough to meet the preceding criteria since the metrics can vary from project to project.

The current list of projects is as follows:

- **OpenTracing** (`https://opentracing.io/`): One of two tracing projects that now come under the CNCF umbrella, the other being Jaeger. Rather than being an application, you download and use it as a set of libraries and APIs that let you build behavioral tracking and monitoring into your microservices-based applications.
- **gRPC** (`https://grpc.io`): Like Kubernetes, gRPC was donated to the CNCF by Google. It is an open source, extendable, and performance-optimized RPC framework, and is already in production at companies such as Netflix, Cisco, and Juniper Networks.
- **CNI** (`https://github.com/containernetworking`): **CNI**, which is short for **Container Networking Interface**, is again not something you download and use. Instead, it is a standard for network interfaces that's designed to be embedded into container runtimes, such as Kubernetes and Mesos. Having a common interface and set of APIs allows more consistent support of advanced network functionality in these runtimes via third-party plugins and extensions.
- **Notary** (`https://github.com/theupdateframework/notary`): This project was originally written by Docker and is an implementation of TUF, which we will cover next. It has been designed to allow developers to sign their container images by giving them a cryptographic tool that provides a mechanism to verify the provenance of their container images and content.
- **TUF** (`https://theupdateframework.github.io`): **The Update Framework** (**TUF**) is a standard that allows software products, via the use of cryptographic keys, to protect themselves during installation and updates. It was developed by the NYU School of Engineering.
- **NATS** (`https://nats.io`): Here, we have a messaging system that has been designed for environments running microservices or architectures supporting IoT devices.
- **Linkerd** (`https://linkerd.io`): Built by Twitter, Linkerd is a service mesh that has been designed to scale and cope with tens of thousands of secure requests per second.
- **Rook** (`https://rook.io`): This focuses on providing an orchestration layer for managing Ceph, Red Hat's distributed storage system, among others on Kubernetes.
- **Harbor** (`https://goharbor.io/`): This is an open source image registry that focuses on security and access with inbuilt imaging scanning, RBAC controls, and image signing. It was originally developed by VMWare.

- **etcd** (`https://etcd.io/`): This is a simple distributed key/value store with a REST API designed to be low latency, and it uses the raft consensus algorithm.
- **Open Policy Agent** (`https://www.openpolicyagent.org/`): This is a unified toolset and framework for policy across your cloud-native stack.
- **cri-o** (`https://cri-o.io/`): This is a lightweight container runtime built for Kubernetes; it allows you to run any OCI-compliant container.
- **Cloudevents** (`https://cloudevents.io/`): This is a specification for describing event data in a common way. There are also SDKs for Go, JavaScript, Java, C#, Ruby, and Python so you can easily introduce the specification in your own projects.
- **Falco** (`https://falco.org/`): Falco provides a cloud-native runtime security engine that parses Linux system calls from the kernel as they are executed.
- **Dragonfly** (`https://d7y.io/en-us/`): This is an open source P2P-based image and file distribution system.

We have used a few of these projects in various chapters of this book, and I am sure that other projects will be of interest to you as you look to solve problems such as routing to your containers and monitoring your application within your environment.

The CNCF landscape

CNCF provides an interactive map of all of the projects managed by them and their members, and can be found at `https://landscape.cncf.io/`. One of the key takeaways is as follows:

You are viewing 1,403 cards with a total of 2,263,137 stars, a market cap of $16.98T, and funding of $66.05B.

While I am sure you will agree that these are some very impressive figures, what is the point of this? Thanks to the work of the CNCF, we have projects, such as Kubernetes, that are providing a standardized set of tools, APIs, and approaches for working across multiple cloud infrastructure providers and also on-premise and bare metal services—providing the building blocks for you to create and deploy your own highly available, scalable, and performant container and microservice applications.

Summary

I hope that this chapter has given you an idea about the next steps you can take in your container journey. One of the things I have found is that while it is easy to simply use these services, you get a lot more out of it by becoming a part of the large, friendly, and welcoming communities of developers and other users, who are just like you, and have sprung up around the various software and projects.

This sense of community and collaboration has been further strengthened by the formation of the Cloud Native Computing Foundation. This has brought together large enterprises who, until just a few years ago, wouldn't have thought about collaborating in public with other enterprises who have been seen as their competitors on large-scale projects.

Assessments

Chapter 1, Docker Overview

Here are some sample answers to the questions presented in this chapter:

1. Docker Hub: `https://hub.docker.com/`
2. `$ docker image pull nginx`
3. The Moby Project
4. Mirantis Inc.
5. `$ docker container help`

Chapter 2, Building Container Images

Here are some sample answers to the questions presented in this chapter:

1. False; it is used to add metadata to the image.
2. You can append `CMD` to `ENTRYPOINT`, but not the other way around.
3. True.
4. Snapshotting a failing container so that you can review it away from your Docker host.
5. The `EXPOSE` instruction exposes the port on the container, but it does not map a port on the host machine.

Chapter 3, Storing and Distributing Images

Here are some sample answers to the questions presented in this chapter:

1. True.
2. This allows you to automatically update your Docker images whenever the upstream Docker image is updated.
3. Yes, they are (as seen in the example in the chapter).
4. True; you are logged in to Docker for Mac and Docker for Windows if you use the command line to log in.
5. You would remove them by name, rather than using the image ID.
6. Port `5000`.

Chapter 4, Managing Containers

Here are some sample answers to the questions presented in this chapter:

1. `-a` or `--all`.
2. False; it is the other way around.
3. When you press *Ctrl* + *C*, you are taken back to your terminal; however, the process that is keeping the container active remains running, as we have detached from the process, rather than having terminated it.
4. False; it spawns a new process within the specified container.
5. You would use the `--network-alias [alias name]` flag.
6. Running `docker volume inspect [volume name]` would give you information on the volume.

Chapter 5, Docker Compose

Here are some sample answers to the questions presented in this chapter:

1. YAML, or YAML Ain't Markup Language.
2. The `restart` flag is the same as the `--restart` flag.
3. False; you can use Docker Compose to build images at runtime.
4. By default, Docker Compose uses the name of the folder that the Docker Compose file is stored in.

5. You use the `-d` flag to start the container's detached mode.
6. Using the `docker-compose config` command will expose any syntax errors within your Docker Compose file.
7. The Docker App bundles your Docker Compose file into a small Docker image, which can be shared via Docker Hub or other registries, the Docker app command-line tool then can render working Docker Compose files from the data contained within the image.

Chapter 6, Docker Machine, Vagrant, and Multipass

Here are some sample answers to the questions presented in this chapter:

1. The `--driver` flag is used.
2. False; it will give you commands. Instead, you need to run `eval $(docker-machine env my-host)`.
3. False; Docker needs to be installed.
4. Packer.
5. The Docker daemon configuration is no longer considered best practice.

Chapter 7, Moving from Linux to Windows Containers

Here are some sample answers to the questions presented in this chapter:

1. You can use Hyper-V isolation to run your container within a minimal hypervisor.
2. The command is `docker inspect -f "{{ .NetworkSettings.Networks.nat.IPAddress }}" [CONTAINER NAME]`.
3. False; there are no differences in the Docker commands that you need to run to manage your Windows containers.

Chapter 8, Clustering with Docker Swarm

Here are some sample answers to the questions presented in this chapter:

1. False; the standalone Docker Swarm is no longer supported or considered a best practice.
2. You need the IP address of your Docker Swarm manager, and also the token that is used to authenticate your workers against your manager.
3. You would use `docker node ls`.
4. You would add the `--pretty` flag.
5. You would use `docker node promote [node name]`.
6. You would run `docker service scale cluster=[x] [service name]`, where `[x]` is the number of containers that you want to scale by.

Chapter 9, Portainer – a GUI for Docker

Here are some sample answers to the questions presented in this chapter:

1. The path is `/var/run/docker.sock`.
2. The port is `9000`.
3. False; applications have their own definitions. You can use Docker Compose when running Docker Swarm and launch a stack.
4. True; all of the stats are shown in real time.

Chapter 10, Running Docker in Public Clouds

Here are some sample answers to the questions presented in this chapter:

1. An Azure web app
2. An EC2 instance
3. False
4. Knative
5. Amazon ECS

Chapter 11, Docker and Kubernetes

Here are some sample answers to the questions presented in this chapter:

1. False; you can always see the images used by Kubernetes.
2. The `docker` and `kube-system` namespaces.
3. You would use `kubectl describe --namespace [NAMESPACE] [POD NAME]`.
4. You would run `kubectl create -f [FILENAME OR URL]`.
5. Port `8001`.
6. It was called Borg.

Chapter 12, Discovering other Kubernetes options

Here are some sample answers to the questions presented in this chapter:

1. False
2. MiniKube, Kind, and K3d
3. MicroK8s and K3s

Chapter 13, Running Kubernetes in Public Clouds

Here are some sample answers to the questions presented in this chapter:

1. `kubectl create namespace sock-shop`
2. `kubectl -n sock-shop describe services front-end-lb`
3. `eksctl`

Chapter 14, Docker Security

Here are some sample answers to the questions presented in this chapter:

1. You would add the `--read-only` flag; or, if you wanted to make a volume read-only, you would add `:ro`.
2. In an ideal world, you would only be running a single process per container.
3. By running the Docker Bench Security application.
4. The socket file for Docker, which can be found at `/var/run/docker.sock`; and also, if your host system is running Systemd, `/usr/lib/systemd.`
5. False; Quay scans both public and private images.

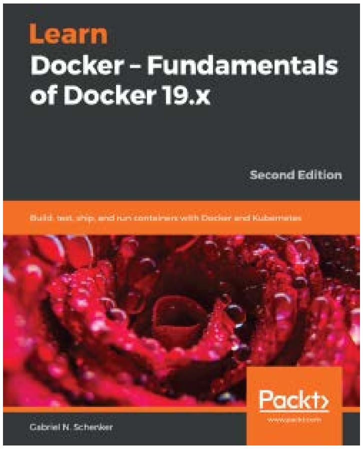
Chapter 15, Docker Workflows

Here are some sample answers to the questions presented in this chapter:

1. The Nginx (web) container serves the website; the WordPress (WordPress) container runs the code that is passed to the Nginx container.
2. The `wp` container runs a single process, which exists once it runs.
3. cAdvisor keeps metrics for only 5 minutes.
4. You would use `docker-compose down --volumes --rmi all`.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

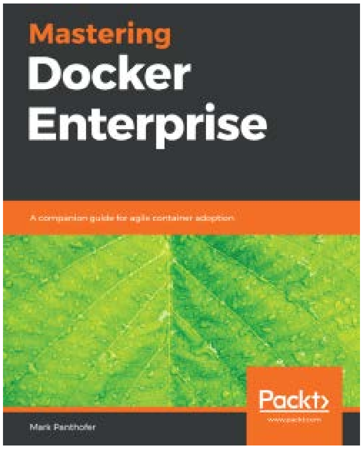

Learn
Docker – Fundamentals of Docker 19.x
Second Edition
Build, test, ship, and run containers with Docker and Kubernetes
Packt>
www.packt.com
Gabriel N. Schenker

Learn Docker - Fundamentals of Docker 19.x - Second Edition

Gabriel N. Schenker

ISBN: 978-1-83882-747-2

- Containerize your traditional or microservice-based applications
- Develop, modify, debug, and test an application running inside a container
- Share or ship your application as an immutable container image
- Build a Docker Swarm and a Kubernetes cluster in the cloud
- Run a highly distributed application using Docker Swarm or Kubernetes
- Update or rollback a distributed application with zero downtime
- Secure your applications with encapsulation, networks, and secrets
- Troubleshoot a containerized, highly distributed application in the cloud



Mastering Docker Enterprise

Mark Panthofer

ISBN: 978-1-78961-207-3

- Understand why containers are important to an enterprise
- Understand the features and components of Docker Enterprise 2
- Find out about the PoC, pilot, and production adoption phases
- Get to know the best practices for installing and operating Docker Enterprise
- Understand what is important for a Docker Enterprise in production
- Run Kubernetes on Docker Enterprise

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index